The Software Developer's Sourcebook

From Concept to Completion

Blaise W. Liffick

Foreword by

Esther Dyson

The Essential Reference

THE SOFTWARE DEVELOPER'S SOURCEBOOK

From Concept to Completion: The Essential Reference

BLAISE W. LIFFICK

Department of Mathematics
and Computer Science
Millersville University
Millersville, Pennsylvania

Addison-Wesley Publishing Company, Inc.

Reading, Massachusetts Menlo Park, California
Don Mills, Ontario Wokingham, England Amsterdam
Sydney Singapore Tokyo Mexico City Bogotá
Santiago San Juan

This book is dedicated to my wonderful wife, Becky, and my fantastic kids, Alexander, Karyn, and Gwendolyn. They sacrificed a normal life so that I could exorcise the writing demon.

Library of Congress Cataloging in Publication Data

Liffick, Blaise W.
The software developer's sourcebook.

Bibliography: p.
Includes index.
1. Electronic digital computers—Programming.
I. Title.
QA76.6.L536 1985 001.64'2 84-28342
ISBN 0-201-11771-1

BCDEFGHIJ-HA-898765
Second Printing, September 1985

Text design by Dana Kasarsky Design, NY, NY
Text set in 11 pt. Baskerville by Techna Type, Inc., York, PA
Cover design by Marshall Henrichs

Contents

Foreword
by Esther Dyson

In the software business, you don't play to win the match. You play to win the round and get a crack at the next. The only way to win the match is to sell out and retire, and that's boring. *The Software Developer's Sourcebook,* part of Addison-Wesley's Software Development Series, will give you guidance as you design and develop your product. Although Blaise Liffick's *Sourcebook* is written for both entrepreneurial and corporate situations, I prefer to address those who consider themselves entrepreneurs, whether in fact or in spirit.

As you attempt to develop your software and build a company or a reputation around your products, you will find that you are competing not so much for people to buy your product as for people to market it. With the right people, you can do anything (almost!). Good products can be bought, but good people who enhance, support, and market those products are necessary to build a company. These people are all you can rely on in the long run, for products become outdated and advertising becomes stale. Only people have the capacity to keep your company winning round after round; only people have the capacity to effect change.

The kinds of people you need are changing in line with the industry itself. The marketplace is also changing, from a world of enthusiasts and tinkerers to a world of sellers, who regard software as an artifact to be developed, polished, produced, promoted, and sold, and of users, who regard software as a tool that can solve their problems. The less the users see of the software, the better, as far as they are concerned; if the computer snatched away the problem and presented the answer, the user would be delighted. The user doesn't care how, or why, it works, as long as it does.

In a sense, it's hard for the creators and tinkers in this industry to accept the fact that their task is becoming subordinate to that of the marketers, financiers, and product planners. But this is *software!* they protest. You can't treat it like soap. But indeed we can; indeed we must.

Over the next few years software will continue to become more of a business and less of a festival. Well-financed, well-organized, giant companies are moving in. The small company with a good product will often be squeezed out by the big company with an adequate product.

So, how do you—presumably a small company with a good product—compete? There are two options: You can decide that you're another Lotus and do things on a grand scale. (If you're really that good you probably don't need this book.) You can get the best venture capitalists, hire the most brilliant marketing people, and get free publicity.

More likely, you've got a good product and need to find its market. You're a small company with a bright future, and more interested in profits and the satisfaction of doing a good job than in appearing on the "Today Show." Your market may be smaller than Lotus's, but so are the resources you'll have to use to reach it.

Indeed, one extremely fortunate fact of life for small software businesses is that software, as a tool, can be designed to solve specific problems. Moreover, as we move toward "end-user computing," we realize that users expect to buy solutions, not general tools. Languages, operating systems, utilities, and spreadsheets are of little interest to end users. They want specific solutions. This fact has a significant impact on the industry's structure.

While segmentation—in the toothpaste industry, for example—is mostly a creation of marketers, segmentation in the software industry reflects reality. In other words, toothpaste may be targeted at smokers (Topol), would-be sexpots (Close-Up), parents of children (Crest), and children themselves (remember Stripe?), but basically all toothpastes perform the same function. The difference is in the users' (and marketers') minds. Toothpaste is fungible. In a pinch, a sexpot could use Crest, or a smoker could use Pepsodent, with perfectly adequate results.

But try doing your payroll with a word processor, or running a bakery with a dental office package. Except for systems software and tools such as Symphony, dBASE III, and the pfs series, software solves specific problems. Only a very few packages actually compete with each other—although these are the ones that attract most of the press's attention.

Thus the micro software marketplace is inherently more hospitable to small companies than, say, the toothpaste industry. While it is every company's ambition to grow larger and more successful, even a small company can compete on relatively equal footing with large companies if it addresses a tightly-defined market. IBM may have billions of dollars to spend on marketing, but you can rest assured that it will spend most of them on hardware and on generic applications that solve a wide variety of general problems but few specific ones. How to reach your best market is the topic of the first book in the Software Development Series, *Marketing Your Software.*

Thus, the trick to success in the software business is not *finding* opportunities but fending them off. There's a constant temptation to say, for example, "We have a great package for lawyers that we could adapt with just a little effort to suit doctors." True, but it will take a lot of extra effort to *market* to those doctors. Another pitfall is the extra machine "opportunity." "Let's rewrite this package for the Yet-Another PC. It's almost identical to the IBM PC." But don't forget the cost of rewriting the documentation to fit the quirks of the Yet-Another and the time needed to support its users. You might find that applying those same resources to your original product would produce much better returns.

There's a lot of talk about the perils of being a one-product company, but the perils of being a two-half-product company are far greater, and, unfortunately, far more prevalent. Launching a second product may be exciting, while improving your original product can be a little dull. Remember, the users and potential buyers of your first product don't care about excitement; they want a good product that works. The reputation of your first product will sell (or hinder sales of) not just your first product but, eventually, your second. So make sure you earn that good reputation before you move on to a second product.

Is the industry losing its romantic, pioneer spirit? Is it addressing such a wide audience that it's losing its special character? Perhaps. But good marketing is not pandering, nor is acceptance of a product by an audience of normal people rather than geniuses a sign of brainlessness. Indeed, in the same vein as Frank Perdue's immortal words, "It takes a tough man to make a tender chicken," it takes a smart programmer to write an easy program.

Intellectual pleasure need not be limited to the arcane world of computer programming. Developing products to fill a specific market—discerning a need, figuring out how to fill it, designing the product to do so, and communicating to the potential buyer what you have done—can bring a lot of satisfaction.

Esther Dyson is publisher of *Computer Industry Daily,* the industry's first serious daily newspaper. Formerly, she was president of EDventure Holdings, Inc., and editor-in-chief of RELease 1.0.

Preface

There is a great secret that is being kept by all of the computer manufacturers, computer stores, and most publishers. That secret is revealed in this book, and comes in two parts.

The first part is: "Programming is not easy." Contrary to the popular belief promoted by most computer manufacturers, it is not possible to learn how to write programs by following their ten easy steps. To compound this sinister plot, the vast majority of introductory books on programming also promote this notion. This is especially true of books that deal with programming in BASIC, which is the major programming language used on microcomputers. The clear statement of such books is that all you have to learn is a few of the BASIC instructions, and you, too, can make big money as a programmer.

In some ways, this is a cruel hoax. It entices parents to buy computers for their kids, because no parent wants to deprive their own flesh and blood of the ultimate opportunity to learn everything that is worth learning. In addition, parents are enticed to try their hands at programming, since it is obviously so easy and canned programs are so expensive. It entices businesses and schools to buy computers and to attempt to develop their own applications after only reading the instruction manual. In this case, people who have enough real work to do are expected to take up the extra load of programming.

This is not to say that these people do not have any *need* to program. Indeed, there are many applications that are specific to a particular business, or have not yet been developed, that these people might create for themselves. It is often helpful to be able to develop short applications to perform some small task.

In addition, many people find programming enjoyable. As a result, they may get "hooked" on programming and wish to pursue it as more than a hobby. These people have genuinely unique applications in mind that could help solve many people's problems. However, they lack the skill of an experienced programmer to bring that application to life. The vast majority of programming texts available are introductory in nature and don't provide enough depth to help this class of programmers. Books that cover programming in more depth are usually written for computer scientists and are much too technical in nature for a new programmer to use.

The second part of the big secret is: "There are many techniques that professional programmers use that make life much easier." The bad news is that most of these techniques are hidden in those highly technical books, not in the introductory ones.

This book, like most technical books, was written in response to a perceived need. That need is obvious from watching many would-be programmers struggle with developing anything more than a trivial program. The problem is that no one has yet told them about the "secret" techniques professional programmers use to develop sophisticated, high-quality programs.

These techniques are not always easy to use, in that they require a fair amount of work. The new programmer simply must not be easily discouraged. Programming is an arduous task, but it can be ultimately quite rewarding. Although it might appear that some of the professional techniques make the programming task more difficult, in reality they make it a much simpler process overall.

This book reveals these "secret" professional techniques. The techniques presented are essentially computer and language independent. However, some examples are given in the Microsoft version of BASIC and in Pascal. The reader is expected to be familiar with programming in some language, but it is not essential that the language be either BASIC or Pascal.

By faithfully following the techniques presented, the new programmer can become experienced enough to develop systems of professional quality and sophistication.

Acknowledgments

I would like to express my thanks to the many individuals who helped make this book a reality. First, I would like to thank the faculty members of the Department of Mathematics and Computer Science at Millersville University for supporting this work by providing sounding boards for ideas. Next, I greatly appreciate the students of CS 625 who "volunteered" to be guinea pigs for much of this material, and who provided valuable feedback. Special thanks to L. T. Thanks to Linda Steinmetz for technical help with balky printer connections and encouraging comments. Finally, much thanks to Ted Buswick, Mark Corsey, and the production crew at Addison-Wesley for their much-appreciated advice and artful work.

B.W.L.

Introduction

Software development (commonly called programming) has been variously described as an art, a science, and an engineering discipline. There is at least some truth in each, and there is no requirement for mutual exclusion among them.

As an art, programming certainly has much of the flavor of something immensely creative, and we have little trouhle envisioning the programmer as an artist who spends endless solitary hours putting words (code) to paper in an order that is not only clever (and perhaps even unique) but is also useable. While perhaps this is more like an interior designer or architect than someone who takes hrush in hand, there is an impression of many of the hrethren being a hit like a Beethoven or Van Gogh, consumed by their work and creating masterpieces seemingly out of nothingness.

Certainly programming can be likened to the creation of a play or novel; while the plot may seem somewhat familiar (i.e., the same topic has been treated by more than one author), the way the writer says things is new, or at least the writer has new insights into the particular topic. Even more in imitation of art, creators of video arcade games are sometimes promoted like rock stars. Home game cartridges are being packaged as elaborately as any record album, including photo spreads and detailed biographies of the programmer. Naturally, it is a moot point whether or not these imitations are indeed art.

Many computer professionals, especially those of us in academe, have been claiming rather loudly that the field is called Computer Science (you can actually hear the capitalization), and that any other moniker is an insult. Programming is definitely an example of the application of the scientific method: recognition and formulation of a problem, observation and experimentation, and finally the formulation and testing of hypotheses.

However, it is in the area of experimentation that computer science is most *unlike* physical sciences. Little can be done to provide experimental data ahout a particular "problem" requiring a computer solution, in the same way that an architect can do little in experimenting with the design of buildings. Certain immutable laws exist that provide boundaries for such experimentation. Theoretical computer science is more in the realm of mathematics, and in fact is a natural outgrowth of what is sometimes called the mathematical sciences. However, as is true of much that is theoretical, there is often little *practical* use for the results obtained in this area.

Programming is probably most strongly akin to engineering. By definition, engineering is an applied field. It is concerned with making structures, machines, products, systems, and processes useful to mankind. In the computer field, the term "software engineering" is being increasingly used to indicate that software development must be primarily concerned with constructing useful tools. Such development requires using techniques, methodologies, and previously constructed building blocks to build new, improved, practical computer tools. Of course, the aspects that make engineering software "applied" are many times at odds with art and science. Certainly, the artist and scientist have a right to balk at any attempt to force their work to be of practical value. However, they must also allow that their work might be employed in a practical manner. And in many senses, so much the better, if such application is appropriate.

Why, then, is the book not entitled *An Introduction to Software Engineering?* First, the term "software engineering" has only recently been coined, and is not yet well defined. Second, the texts currently available on software engineering are written at the junior or senior computer science major level, and therefore assume a significant background in computer science topics. Third, there are a number of topics that are related to software development but are only on the periphery of software engineering.

A fanciful subtitle for this book might be *A Holistic Approach to Top-Down, Bottom-Up, Modular, Structured, Human-Engineered Programming for Teachers, Students, Small Business Users, Home Computerists, and Other Followers of the Grail.* Holism is defined by Webster's Collegiate Dictionary as "a theory that the universe . . . is correctly seen in terms of interacting wholes . . . that are more than the mere sum of elementary particles." Paraphrased, this reduces to "the whole is greater than the sum of its parts." The only problem in applying the term holism to the computer field lies in its popular image as somehow connoting mysticism or a Zen-like philosophy. This is unfortunate because the term holism correctly used can help to describe a more global approach to the complex subject of computer science, which is, at best, a spaghetti bowl of seemingly unrelated topics.

The parts to be summed are the techniques used during the various stages of software development. Each technique has its own unique set of adherents. Yet what is usually ignored is that these techniques are not mutually exclusive, but in fact can be complementary in the total development effort. While you would not necessarily use every technique discussed while developing a particular piece of software, it is rarely made clear, even to professional programmers, that it is perfectly all right to do so. Because current books tend to deal, at most, with a couple of the topics to be discussed here, it is nearly impossible for anyone other than a computer scientist to find appropriate reference material for making life easier for the software developer. In addition, what references are available typically require that the user have a background equivalent to a BS degree in computer science in order to understand them.

This book is intended more for the "amateur," the nonprofessional programmer. This means that the reader's primary responsibility is not to write programs hut is something such as managing, teaching, or homemaking. But just because these individuals are not professional programmers doesn't mean that they won't write programs that require sophisticated techniques.

As a result of the explosion of the microcomputer market, enormous amounts of raw computer power are in the hands of non–computer specialists. These "uninitiated souls" are encouraged, even enticed, to write programs for their own use. However, the hardware manufacturers, software vendors, and publishers have not provided adequate means for these individuals to actually develop sound, quality, useful software. These novices are being led to believe that all they need to be concerned about is learning how a few BASIC statements work, and the program will practically write itself! It's like teaching you how to bait the hook while ignoring the details about how you actually catch a fish.

This book will play several important roles for the reader. First, it acts as a primer to the various techniques available to help make life simpler as a programmer. Second, it makes the reader a more knowledgeable consumer of software by providing effective yardsticks by which to measure other people's products. Third, since in many ways the topics discussed are "timeless" and are not in any way based on computer *technology*, this book will serve as an excellent reference tool for many years to come.

As noted, this book is machine independent, and need not be applied solely to the microcomputer field. The majority of the topics discussed are as important to programmers using the largest mainframe systems. In fact, several of the techniques have been "borrowed" from this environment, and are heavily used by computer professionals. However, several of the topics are more easily demonstrated assuming microcomputer-based systems, and so it will be assumed that the reader will be comfortable in this type of programming environment.

In addition, many of the examples, especially in chapters 3, 4, and 5, will use BASIC. The format of the code will be as general as possible so that the examples apply to most microcomputer systems available. However, the code presented will be in Microsoft BASIC, making it directly applicable to the IBM PC, the DEC Rainbow 100, and any CP/M-based system. There will also be a great deal of compatibility with the Apple II line of computers. While the details of how a particular technique is implemented in BASIC differs from machine to machine, the fundamental concepts being applied do not change. Therefore, users of other dialects of BASIC should he able to apply the techniques in this book as well.

In addition, users of other programming languages such as Pascal or COBOL should find this book helpful. In fact, much of this material may be easier if you are using another language. This is especially true of Pascal, since the design of the language directly incorporates many of the concepts presented. The topics discussed are therefore largely hoth machine and language independent.

1 Systems Analysis

1·1 INTRODUCTION

The first impulse of most inexperienced programmers is to begin writing code as soon as they have an idea for a program. This impulse is stronger since the advent of microcomputers and **interactive** programming, where the programmer can get immediate feedback from the program. Indeed, one of the driving purposes behind the development of BASIC was to create a programming language that gave feedback as soon as possible during the programming process.

Unfortunately, bowing to this impulse leads to a trial-and-error method of programming that increases the difficulty of the task considerably. Only recently have "amateur" programmers, i.e., persons whose main responsibility is not programming, begun to discover what professional programmers have known for many years: the task of programming requires a disciplined, methodical, and well-orchestrated technique in order to assure success.

Application programs are not created in isolation. It is first necessary to recognize that the development of applications is driven by the needs of potential users. In a large company, the typical scenario is that a user will discuss his needs with the computer support staff. Once this discussion has taken place, the computer staff may define and develop a new application. Next, programmers will implement this application. Finally, the user will be set up with some procedure for accessing it.

What Is a System?

The term "system" has several definitions in the computer field. First, it can mean the hardware itself, as in "computer system." The relationship between the parts of this system is typically as simple as a physical connection. In this sense, the computer and its peripherals represent a system. Together, these individual pieces perform a collective function.

The term "system" can also be used to refer to software, as a synonym for "application program." Its use in this sense is often reserved for very complex programs, however.

A third meaning for the term, again referring to software, is found in the reference to a "system of programs." In this case, there are individual programs that collectively accomplish some task, although a user might not be able to easily distinguish the individual programs within the system.

Finally, "system" can refer to a combination of hardware and software that together perform a specific function. An example of this is what is usually called a **turn-key** system, where the computer is set up with a particular application that begins to execute as soon as the computer is turned on. In this way computers are specialized to run a particular application package.

The most general definition for "system" might be "a series of interrelated elements that perform some activity, function, or operation." [96] In this case, there is no direct connection with computer technology. Indeed, every office has a number of "systems" that operate on human, not computer, power. A complete system might even include some actions performed by people as well as by a computer.

What Is a Systems Analyst?

Systems analysis is the study of the relationships and interactions of the various components of systems. Considering the more general definition of the term "system," a **systems analyst** does not necessarily have anything to do with computers, as in the case where the system being studied is entirely manual.

Why does a systems analyst study systems? What is the end result of such analyses?

An early example of this type of work was the efficiency expert, someone who would observe the functioning of an office or manufacturing environment in order to detect methods that were inefficient. The object of the analysis was to improve efficiency. This was not, in itself, the final goal, however. The hope was that by increasing the efficiency of the operation, there would be a related increase in productivity. This increased productivity would result in a lower overhead cost for the operation, either because the same operation could be accomplished with fewer employees, or because the same number of employees could produce more. The bottom line, thus, was a higher profit.

Today, the term "systems analyst" is mainly associated with the computer field. While the analyst may study systems that are essentially manual, the purpose of such study is to determine whether the system can or should be computerized. The ultimate goal is still the same, increased productivity and thus profit. The operating theory is that a computerized system is invariably faster, cheaper, and less error-prone. While this is occasionally debatable, any system that can be, usually is automated eventually.

The Analyst's Role: Part 1

The analyst really has two main jobs. First, it is the analyst's responsibility to determine, in great detail, exactly how the present system functions. This is seldom a quick or easy task. Many of the procedures being followed by the various staff involved in an operation may not be formally described in any document. "Oral tradition" still plays a great role in how things get done in most companies.

Another problem is that the procedures may not be well understood by those following them. They may not know *why* they do things the way they do, only that "it's the way things have always been done." But the analyst must discover the *why* for every step in the operation so that its relative merit can be determined. It could be that some steps are out-of-date or meaningless. It is difficult to know whether a step is meaningless if you don't know why it is being done, however.

Next, it may be difficult to discover what all of the steps in the operation are, even if there is some documentation for the operation. If a particular staff member is out of the office the day the analyst interviews everyone, important details might be missed. It is even possible that a necessary part of the overall system being studied is not discussed with the analyst because it is not thought of as part of the operation.

All of these problems make the determination of how an existing system works often the most difficult part of the analyst's job. The analyst must act more than a little like an investigative reporter, ferreting out important information from the most unlikely places. Only after all of the procedures for the present system are understood and described in detail can any attempt be made to computerize the system.

There are three main ways that the analyst gathers pertinent information. First, any documentation that exists for the current system must be studied in detail. This includes any manuals describing procedures, tables that describe how certain mechanisms work, examples of all forms used in every step (no matter how infrequently), and any "crib sheets" that individuals may have prepared for their own portion of the system.

Second, the analyst must interview every employee who has a significant role in the operation of the current system. Exactly what "significant" means is often difficult to pinpoint. The safe bet is to interview everyone involved, but this could be extremely time-consuming. Where the system involved is very large, very complex, or involves many personnel, it is often necessary to have a team of analysts.

Interviewing so that the maximum amount of information is gained in the minimum amount of time is not an easy procedure. How a question is posed can be as important as the question itself. And since the analyst is dealing with people, not all of whom are going to necessarily be completely cooperative, the analyst must be able to handle the intricacies of human interactions. This is many times the most difficult part of the entire procedure. (The reader is referred to the Bibliography for further information on the various methods for formal interviewing.)

The final information gathering method employed by the analyst is observation. Many times, the analyst can more easily determine what is being done by simply observing the various procedures in progress than by any other means. While it is difficult to be certain that some subtle actions are not missed by the analyst using this method, direct observation is still a useful tool for verifying information obtained in other ways, if nothing else.

The analyst is collecting information about the system in an attempt to answer five short, but deep, questions:

1. What is currently being done?
2. Why is it being done?
3. Who is doing it?
4. How is it being done?
5. What are the problems involved in the current method?

Once these questions have been answered in detail, a description of the current system emerges. It is this description that will form the basis for any computer application that is to replace the current system.

The Analyst's Role: Part 2

The second part of the analyst's job is to describe a computerized replacement for the current system. This is the part that is most often associated with a systems analyst, since it deals more directly with computer applications.

The analyst has several tasks at this stage. First, the analyst must prepare a description of the new computer application based upon the description of the current system just analyzed. This begins as an effort to describe how the new application will resolve the problems associated with the current system. In addition, however, the new application should take advantage of new ways of accomplishing the tasks at hand, even to the point of changing the procedures of the system significantly. The analyst must recognize these opportunities to improve the system.

The analyst will also begin describing the system as a series of sub-tasks to be accomplished in some order. This was perhaps already done to an extent in the original system because multiple personnel were involved, each with his or her own procedures to follow. The analyst must now define logical sub-tasks that a computer can follow, perhaps regrouping those originally present.

The analyst's next responsibility is to anticipate the automation of the task. This creates two main concerns. First, since the way in which data is maintained and handled can greatly influence how efficient a system is, and because the ultimate goal of any system is to maintain and produce data in various forms, the analyst must be aware of what organization of the system's data would be most appropriate and effective.

Second, the analyst must anticipate what problems the computerization of the system might cause. Perhaps certain procedures in the system are time-sensitive. For instance, it occasionally creates a problem when a computerized application performs certain steps of a procedure too quickly. A delay introduced because of the inherent time it takes for a human clerk to enter an order into an order book may be essential for the maintenance of a proper inventory level. Speeding up the ordering process by computerizing it may cause the inventory to be suddenly depleted.

The final responsibility of the analyst is to act as a communications link between the technology and the end users, who typically know little about computers. As a liaison, the analyst helps the user to understand the new application. In addition, the analyst acts as the user's advocate with the computer department. He or she helps users to explain their needs and protects the user's interests during the formulation of a new application. A user should not have to accept an awkward procedure simply because the computer staff wants to implement an application in a certain way.

Applications from Scratch

It often occurs that someone gets an idea for a computer application that has never existed as a manual system in any significant sense. For instance, imagine getting an idea for a new computer game. Since the initial description of the new application is typically based on the analysis of some system currently being used, how does one begin to describe such an application?

Since there is no current system, the initial analysis phase that was described above is obviously impossible. However, you want to end up with the same type of system that you would have if the analysis were possible. This is accomplished by trying to answer the same questions that you would answer if the system had actually existed already. This requires a lot of imagination and a certain amount of trial and error until the description you arrive at is satisfying.

The System Life Cycle

A **system life cycle** is the set of steps required to take a computer application from conception to finished product, whether the project is a single program or a very large system of intricately interconnected programs. Unfortunately, there is no accepted standard for defining the life cycle of a programming project. This is no different from there being no standard method of company management. The life cycle is a method for managing the programming project. Since the requirements of individual projects can differ widely, the details of the life cycle can also differ.

Many life cycles include elaborate schemes for including feasibility studies, formal proposals, and detailed reviews. While these steps are often necessary when dealing with a large corporation or major project, they are seldom required for the amateur programmer. As a result, these additional steps will be ignored in the system life cycle defined here.

Definition Phase

The place to start with all projects of this sort is at the **definition phase.** This phase, like the others that will be discussed, can be as elaborate or as informal as you wish. For personal projects, I typically try to hit a middle ground between the two extremes.

In this phase, we are concerned with defining what the application is. As discussed above, this includes the general analysis of a system. This phase also includes the **preliminary design.**

The object of the preliminary design is to provide a general outline for the system design. It is best to leave until later as many details as possible about *how* a system is going to perform its functions. As will be seen, this promotes many sound habits that will improve the quality of the programs you develop. The preliminary design portion of the definition phase overlaps a bit with the **design phase.** This results from the methods used in analyzing a system, which can contribute a great deal to the preliminary design. The term "preliminary design" encompasses these tools in much of the literature, since the term is used informally in the computer field.

Design Phase

The **detailed design** of a system is a more specific design description of the program's logic and functional components. While still avoiding issues related to a specific machine or programming language, this phase is the first presentation of *how* the program will perform its functions. However, this is still a general description in outline form. This allows the logic of the program to be mapped out without a commitment to a specific machine or programming language, making it possible to develop the program for many different computers or in many different languages without starting from scratch every time. Even when this is not a specific goal for a project, it is still a vital step in the life cycle of any program.

This design phase is much like the development of an architect's blueprints. The preliminary design corresponds to the first rough sketches and models that an architect makes. Eventually, these sketches are drawn in some final form, which corresponds to the general description of the program. Next, the blueprints are developed from the architect's drawings.

It is these blueprints that make constructing a building truly possible. No construction person in his right mind would begin to put up a building without such blueprints. They describe in detail *how* the building is to be constructed. The architect's job is to see that the building adheres to sound architectural practices, taking into account the physics behind materials and construction techniques. In a similar way, a systems analyst prepares a detailed design of a program, adhering to sound programming practices.

Implementation Phase

After the program has been fully designed, the **implementation phase** begins. It is finally time to become concerned with all the machine and language details that have been avoided so scrupulously thus far. During this phase, the design is turned into code.

If the design was well developed, this phase is somewhat mechanical. In most larger companies, it is at this point that the project is turned over to a programmer by the analyst. Some companies do this during the previous phase, prior to the detailed design. Exactly when this is done is usually a matter of company policy, often depending on the exact nature of the life cycle the company follows.

Verification Phase

Once the program has been coded, it is necessary to evaluate how well it performs. This **verification phase** is often the most difficult and time-consuming step, since it is here that errors must be found and corrected. Many times these **bugs,** as they are known in computer parlance, are quite difficult to detect.

While this phase is usually thought of as following the implementation phase, it really comes into play after both the design and implementation phases. One purpose for doing the design in a general form is so that it can be evaluated. This evaluation includes some preliminary testing for errors, as well as a verification by the potential users that the system being developed will meet their needs. This is not always straightforward or simple, but such a verification can increase the quality of a product immensely, not to mention creating contented users.

Documentation Phase

Once the system has been fully implemented, it still cannot be released for use until detailed documents are created that explain how to use the system. This **documentation phase** is similar to the verification phase, in that documentation is ideally prepared at several points during the life cycle, not only at the end.

This phase is usually the most dreaded by programmers, for several reasons. First, there is a general fear of writing among programmers, which is somewhat odd considering how akin to creative writing the development of programs really is. Second, writing documentation obviously is not as much fun as writing code. Third, and perhaps in many ways most important, company management usually places little emphasis on documentation. The biggest hazard of leaving the documentation task to the end is that the programmer's manager will give the programmer a new assignment as soon as the previous project passes the verification phase. This leaves the programmer little time to create the necessary documentation. As a result of these problems, most systems are vastly under-documented.

Production and Maintenance Phase

The last phase has two parts. A program enters **production phase** when it is finally made available to users for real work. There is little that the programmer needs to do during this phase for a production system that is working well. For larger systems, it is sometimes the programmer's responsibility to set up special executions of the program, such as at the end of a predetermined period or at year's end.

The **maintenance phase** overlaps with the production phase. The real task of a programmer during this time is to monitor the program's use. Changes (i.e., maintenance) will have to be made to the system on occasion. There are two main reasons for this. First, the requirements of the user may change. This may mean adding, deleting, or changing functions in the program to suit the users' new requirements. Second, bugs are invariably discovered (usually by the user). Such errors must be diagnosed and corrected. The system must then be retested. Once this has been done, the new version of the system can replace the current version in production.

1·2 DEFINING THE APPLICATION

While, in many ways, creating a general description of an application is the easiest step of the entire process, it is also one of the most important. If the original description is poorly prepared or wrong, the deficiency will quickly become embedded in the system. If such an error is not discovered until several steps later, it is vastly more difficult to correct. As a result, most professionals spend what may seem like an inordinate amount of time getting the general description right.

This is accomplished through an iterative refinement process. During this process, the analyst will make presentations about the definition of the system to the users who requested the application. Although most technical details about a computerized system will be beyond the users, such briefings are used to look for misunderstandings of the users' needs. In addition, these **walk-throughs** familiarize users with new ways to accomplish old tasks using the computerized system.

The users may not like or understand everything that the analyst presents. This usually requires the analyst to redefine portions of the system. After such a redefinition, another meeting of the analyst and users provides a new opportunity to review the system definition. This cycle continues until both the analyst and the users agree with the definition. Such an agreement does not always come easily, however. Both sides must be willing to compromise occasionally in order to get the project into the development phase. Since most professional projects are bound by both budget and time constraints, such compromises can be quite important.

In this section, the four main parts of an application's formal definition will be examined. First, the results of the analysis of any currently used system must be reviewed. Second, the general description of an application will be discussed. While this may initially be based upon the analysis of a current system, this does not necessarily have to be the case. The description could be a heavily modified form of the current system. As mentioned earlier, it could also be the case that there is no system currently being used, or that the one being used is so primitive as to not provide a solid base for a new application. In such an event, the description will have to be based upon an imaginary system.

Third, it is extremely useful to describe what the expected inputs to the system will be. By describing these inputs, we have the beginning point of the system. In addition, we can determine how this application might connect with other programs or systems. For instance, one of the inputs to this application might have been an output from some other system.

Finally, we should describe the set of outputs that the system will generate. This defines the final goal of the application. In this way, we have a firm target in front of us as we continue to develop the application.

Note, however, that we have not in any way indicated *how* the inputs will be transformed into outputs. This level of detail is premature and would only divert our attention from the need to arrive at a comprehensive description of *what* the application is supposed to do.

Analysis of the Current System

In the introduction to this chapter, we looked at the role of a systems analyst. When studying an existing system in preparation for its computerization, the analyst must discover exactly how the current system operates. In addition, he or she must look for the weaknesses of such a system. Finally, in proposing a computerized system, he or she must develop ways to overcome the weaknesses that have been discovered.

Usually the weaknesses are somewhat obvious and are what prompted the analysis in the first place. The two most obvious weaknesses in systems are that they take too long to perform their function, or that they cost too much. The difficulty in systems analysis is that often what appears to be a flaw of the system is only a symptom of a more subtle problem. For instance, both of the previously mentioned problems could be a result of a high rate of errors. In a manufacturing system, for example, this would be evident from a large number of low-quality parts being manufactured. So what started out as a problem of high overhead turns out to be one of quality control.

An Example

People learn best by doing, so let's look at an example that is well-known yet confuses a very large percentage of the population on a monthly basis—balancing the family checkbook. We'll look at the entire system of which the checkbook itself is but a small part. The application we have in mind, however, is one that will make balancing the monthly bank statement much more painless. We will carry this example through some of the other sections of this chapter as well.

First, even if this application is being developed for your own use, you must proceed as if it were for someone else. It is also helpful later on to approach the subject being analyzed as if you knew nothing about it. You should write down every procedure in detail, no matter how trivial it seems. Make as few assumptions as possible, and avoid the temptation to say to yourself, "I don't need to write that down; it's obvious." You can become quickly overwhelmed with all the details that need attention in even the smallest system, so leave nothing to chance and write it all down.

For this application, start by asking yourself questions about how the present system works. This should lead you to a set of procedures that help define the system. For the checkbook system, you might start with the following steps:

1. You write a check and send it to your creditor.
2. Your creditor receives the check, and deposits it in his or her bank account.
3. Your creditor's bank contacts your bank and arranges a transfer of funds, which will be added to your creditor's account.
4. Your bank deducts the amount of the check from your account and transfers that amount to your creditor's bank.
5. At the end of each month, your bank returns to you all the checks cashed since the last statement. It also includes a statement for your account, listing various details about your account, including what the bank considers to be your current balance.

At this point, you are supposed to take the statement provided by the bank and "balance" your checkbook. This balancing is actually a process to determine whether or not either you or the bank has made an error in keeping the books on your account. Since the bank uses a highly sophisticated computer system, it is usually safe to assume that any mistakes were made by the bank customer.

Before looking at the procedure used to balance the monthly statement, let's first examine what types of errors might have been made by the "user." The most obvious errors are of the simple arithmetic type. These are primarily simple subtraction or addition errors, but can also occur if a check value is added to the current balance instead of subtracted. Still other, more subtle errors occur when the wrong value is entered into the check journal. While the arithmetic may be correct, the result is still wrong since the values being added or subtracted are wrong. Finally, errors of omission occur when the user forgets to write a check into the journal.

These weaknesses may form the base of our new application. In other words, we would like our new computerized checkbook system to help overcome these deficiencies in the current system. It is not clear how this objective will be met, but such a revelation is not yet necessary.

Next, let's look at the various forms that need to be dealt with in a checking account. The three main forms are the check itself, the check register, and the monthly statement from the bank.

The check is of little use to us once it has been cashed. Some banks have even stopped returning canceled checks each month, preferring instead to keep a copy of each check on microfilm at the bank. However, in cases of a dispute with the bank or a company about proper payment, the canceled check is still the easiest method of resolving the issue.

The information on a typical check tells to whom the check was written, the amount of the check, the date it was issued, and the signature of the check writer. In addition, most checks provide a blank space that can be used to indicate what the check was written for. Using this space is strictly voluntary, however, while all the other information is required in order to make the check legal. Unfortunately, since use of the space is voluntary, most people don't bother with it, even though it could provide useful information later, for instance, at tax time.

One other piece of information that can usually be acquired from a canceled check is the date it was cashed. This is usually stamped on either the front or the back of the check. In fact, this stamp is the only proof that the check has indeed been cashed. While not a vital piece of information, it again has its uses.

Finally, each check has a unique check number. This is vital, in that it is the only way that the bank tells checks apart.

The check register is what the account holder uses to keep track of the account on a day-to-day basis. Each time a check is written, the check writer is supposed to enter certain information into the register. This information generally includes the check number, the date the check was written, to whom the check was written, and the amount of the check. The account holder is supposed to keep a running balance by subtracting the amount of the check from the previous balance, yielding a new running balance. This running balance is *not* an indication of how much money is in the bank account at the moment the check was written, since checks written previously and already subtracted in the register may not actually have been cashed yet. It is instead an indication of the remaining resources of the account that can be drawn upon without "overdrawing" the account.

In addition to check information, any deposits made to the account are noted in the register. The amount of the deposit is then added to the running balance. The only other information normally kept for deposits is the date of the deposit.

Finally, the monthly statement can vary greatly from bank to bank. It generally includes a list of all checks cashed since last month's statement. This list is usually arranged by date instead of by check number. In addition, a running daily balance for the account is given. This is the actual amount that the bank claims is in the account each day during the statement period. This will seldom jibe with any amount you list in your check register, however. This is because there are usually a number of "outstanding" checks, i.e., checks not yet cashed.

To balance the checkbook, you generally do the following:

1. Add up the values of all the outstanding checks.
2. Add up the values of all the deposits made after the ending date of the statement, i.e., any deposits you made that did not show up on the current or previous statements.
3. Take the final daily balance given on the statement, subtract the total of the outstanding checks, then add the total of the outstanding deposits. The result should equal the last balance written in your check register.

This procedure is fairly straightforward. It requires doing only simple arithmetic. Yet there are probably very few checkbooks in this country that are "balanced." Why? And how do we tell if a checkbook isn't balanced?

In step three above, we indicated that the result should equal the last balance written in the check register. If it does, then the checkbook is balanced, i.e., the bank thinks you have as much money in the account as you do. If the values are not the same, however, there is a problem. But what kind of problem?

As noted earlier, there are a number of possible sources of error. To make matters worse, there may have been more than one mistake made. This can greatly complicate the search.

Since the purpose of this entire exercise is to indicate whether or not an error has been made in keeping the checkbook (i.e., in the account holder's keeping of the check register), it is important to enumerate as much as is practical all the possible errors that might have occurred.

There could be three sources of error in this system. First, the values in the check register could be wrong. Second, the procedure for balancing the checkbook could have been done improperly. Finally, although the possibility is quite remote, the bank could have made an error on the monthly statement.

We have already outlined, on page 13, various possibilities for errors made in the check register. The following errors could have been made during the balancing procedure:

1. Values for outstanding checks or deposits could have been incorrectly copied.
2. A check that was counted as outstanding is not, in fact, outstanding.
3. A check that really is outstanding was overlooked and, therefore, was not added into the outstanding check total.
4. A deposit that was counted as outstanding is not, in fact, outstanding.
5. A deposit that really is outstanding was overlooked and, therefore, was not added into the outstanding deposit total.
6. The value for the bank's last daily balance was copied incorrectly.
7. The totalling of the outstanding checks was done incorrectly.
8. The totalling of the outstanding deposits was done incorrectly.
9. The final computation was performed incorrectly.

This summarizes the procedure of balancing the checkbook pretty well, right? It seems to cover all the essential details of the mechanism. The description even includes lists of where things might go wrong in the system, which could lead to a description of the purpose for a computerized application.

The next step in this analysis is to make certain that every term that has been used is easily defined and completely understood. Most items, such as "check amount" and "final daily balance," are self-explanatory or are readily identified by the forms being used. What's missing?

What is the definition of an outstanding check? Earlier, we used a loose definition related to whether or not the check had been cashed. However, this presupposes that we are in some way keeping track of which checks have been cashed. It is necessary to have a specific mechanism for this, since there is no way to determine the order in which checks will be cashed, or how quickly or slowly they will be cashed. Some checks will be cashed the day they are written, while others may not be cashed for six months or more. *Any* check that has not been cashed, not just those written in the last month and not cashed, must be considered as outstanding.

The check register typically has an additional field for keeping track of outstanding checks. When a check has been cashed, as evidenced by the check being returned by the bank, this field is marked. Any check that is not marked this way is currently outstanding. Outstanding deposits are handled in a similar fashion.

This adds to the list of errors that might be made as follows:

1. A check that should not have been marked as cashed was, in fact, marked.
2. A check that should have been marked as cashed was not marked.
3. A check that was marked as cashed was incorrectly added into the list of outstanding checks.
4. A check that was not marked was omitted from the list of outstanding checks.

A similar list of errors would apply to deposits.

The Application Description

The particular application that will be developed based upon the above analysis is entirely dependent on the goals of the user(s). In this example, there is a wide range of possible applications that could now be described, from the nearly trivial to the quite complex.

A TRIVIAL APPLICATION A significant portion of errors made in the balancing procedure are simple arithmetic errors made during the totalling of outstanding checks and deposits and the final computation of the balance. A user might ask for a program that will help eliminate these errors. The program might be described as follows:

Checkbook Program 1:
The program will assist a user in balancing his or her checkbook. The program will add up the list of outstanding checks and the list of outstanding deposits, which will be provided by the user. After the program is given the final balance listed in the monthly bank statement by the user, it will output a number that should correspond to the final balance listed in the user's check register. If the amounts are equal, then the checkbook register is correct, and the account is "balanced." If the amounts are not equal, there may be an error in the check register. The user must locate and correct the error, and then run this program again to balance the checkbook.

This description is, perhaps, overly specific, in that it gets uncomfortably close to describing details about how the program should work. Details about possible inputs, outputs, and decisions that the program will make should be avoided at this stage as much as possible. However, as we will discuss in the next section, it is often necessary to be somewhat specific at this stage since the user must sometimes tell how something is to be done in order to describe *what* must be done.

As long as the description is kept as general as possible, there is no harm in this. Note, for instance, that the description does not give information on *how* the new balance is to be calculated. This was left out intentionally.

The goal of this program is to produce a value that can be compared to the last balance in the check register in order to determine if an error was made in the register. A completely general description might present this information alone, leaving the other details of the inputs and outputs until later. For instance:

This program will help a user balance his or her checkbook. With the proper inputs, the program will produce a value that can be compared with the last entry in the user's check register to determine whether or not the checkbook is balanced.

This description can be fleshed out once the details of the implementation are known. However, it summarizes the purpose of the program adequately.

A MORE COMPLEX APPLICATION Although simple arithmetic errors are perhaps the most common ones committed while balancing a checkbook, other errors can make the balancing procedure even more time-consuming. The second most common error has to do with what checks and deposits are listed as outstanding. Looking for outstanding checks can be quite difficult when dealing with more than one check register, or with checks that have not been cashed for many months. In addition to the elimination of the simple arithmetic errors, a computerized application could help to eliminate these additional errors by keeping track of the checks themselves.

Checkbook Program 2:
The program will help the user to maintain his or her checkbook and to balance the checkbook automatically at the end of each month. The program will keep track of all checks and deposits of the account. With this information, each month's bank statement can be balanced simply by entering the final daily balance listed on the statement. The program will generate a new balance, which is then compared to the final balance in the user's check register. If these values are the same, then the checkbook is balanced. If the values are not the same, then the user has made a mistake in entering information about the checks or deposits.

It should be obvious that this application calls for a much more sophisticated method of keeping track of checks and deposits than the previous example did. The implication is that the program will keep track of checks and deposits in a way that is similar to the check register, since the program must be able to identify outstanding checks and deposits in order to generate the new balance correctly. This is certainly not a trivial matter, and will require sophisticated use of files. This is implied from the need of the program to keep track of *every check and deposit*, from the very beginning of the account. Such a large amount of information can only be stored in a file.

In addition, what if the user makes a mistake in entering a check's information? It was mentioned in the program description that this is the only type of error that might still occur with this system. Some mechanism must be created in the program for changing the information entered for checks and deposits. This is also implied by the need to change a check's status from outstanding to cashed. In the check register this is done by placing a mark next to the check when it has been returned by the bank. This information must also be available to the program.

Finally, it should be noted that this program does not entirely eliminate the type of error that it was supposed to. It is still possible for the user to incorrectly mark a check as cashed, resulting in an incorrect new balance. In addition, it does nothing to eliminate an error in a check's amount once entered into the system. For this application, there is some question whether the time spent entering the information from the checks and deposits does not outweigh any time savings from the balancing function of the program.

A SOPHISTICATED APPLICATION Even if the application described as Checkbook Program 2 is not worth doing because it doesn't really save the user any time or increase the accuracy of the procedure significantly, there may be another reason to go to such great lengths in a checkbook system. In addition to its function as a checkbook balancer, such a system could form the foundation of a sophisticated record-keeping system, which could be useful in organizing the information for other purposes. For example, if the checks were categorized by their purpose, such as medical expenses, office expenses, etc., the system could generate reports that would simplify the annual chore of filing taxes.

Checkbook Program 3:
This program will be a complete bookkeeping system. It will keep track of all checks written and deposits made by the user. The program will then balance the user's checkbook each month. In addition, the program will provide annual reports for the user, which will list checks that can be deducted in each deductible category for filing federal taxes.

As in the previous program description, this description implies a sophisticated file system. The program must keep track of all checks and deposits as in the previous system. In addition, it must keep track of which tax category each check is in. This information must initially be provided by the user. The program will then be able to use this categorization to produce annual summary reports for preparing taxes.

An application could even be carried one step further by including a program that would help the user to fill out a standard set of tax forms, based upon the information kept by the above application and some additional information provided by the user (such as gross income). It should be obvious by now that the functions of an application can quickly escalate into a very complex system. The best approach is to start out with the most minimal program that will perform the function needed. Additional functions can be added as time permits, and the application can be built up gradually to become a very sophisticated system.

Inputs and Outputs

You probably noticed that many of the descriptions given above dealt with inputs and outputs. This is impossible to avoid, since a program's goals are identified by its outputs, and the outputs cannot be generated without some idea of what the inputs are.

The outputs a program should generate form a target for the development of a program. Think of the computer and its software as a black box. A black box is a box that cannot be opened by the user. It accepts a certain type of input, and generates a predefined type of output. An example of a black box might be a television. The input might be signals received on an antenna. The output is the picture that the user sees. The user does not need to know how the insides of a television work, but must only know that certain buttons on the set, such as the channel selector, control what is seen.

A user wants a black box that will perform the task he or she needs done. When explaining his or her needs to an analyst, the user will usually say something like, "I want a black box that will do *this*." For each user, *this* is a description of the results the user expects from the black box. For instance, the user might say, "I want a black box that will help me balance my checkbook by doing the calculations for me." Another might say, "I want a black box that will calculate the standard deviation of a set of numbers." A third might say, "I want a black box that will help me fill out my tax forms." In the professional programming world, the most often heard request is "I want a black box that gives a report that tells me *this*," where *this* is some arcane statistic.

The only thing that the user really knows about the black box is what the results from it will be. These expected results form the goals for the program that will create the user's black box using a computer.

Therefore, it is usually natural to begin describing a program by what the expected outputs will be. This might include isolated values, such as the checkbook balancer's new balance, or complex reports, such as in the tax system. If the manual system generates specific forms and reports, chances are the new application will need to generate these same forms and reports in some fashion.

Once a general description of the expected outputs has been established, it is usually necessary to begin describing what inputs will be needed to generate those outputs. At this point, the exact nature and format of the inputs is not important. Just knowing that the system must keep track of checks is sufficient.

It is not necessary to describe *all* inputs or outputs now. The general description will provide a gross image of the target and the beginning point. The design developed in the next phase describes the details of how the imputs will be converted into the expected outputs.

Example 1: A Carpet Store Estimator

A friend who owns a carpet store comes to you one day with a problem he would like you to resolve. It seems that his salespeople are having difficulties preparing quick, accurate estimates for their customers. His request is prompted by two things. First, estimates prepared by the sales staff are prone to simple arithmetic errors, since the calculation of the total cost to carpet a room requires converting room dimensions given in feet to square yards, in addition to multiplying and adding many factors. Even though the salespeople use calculators, they make too many mistakes.

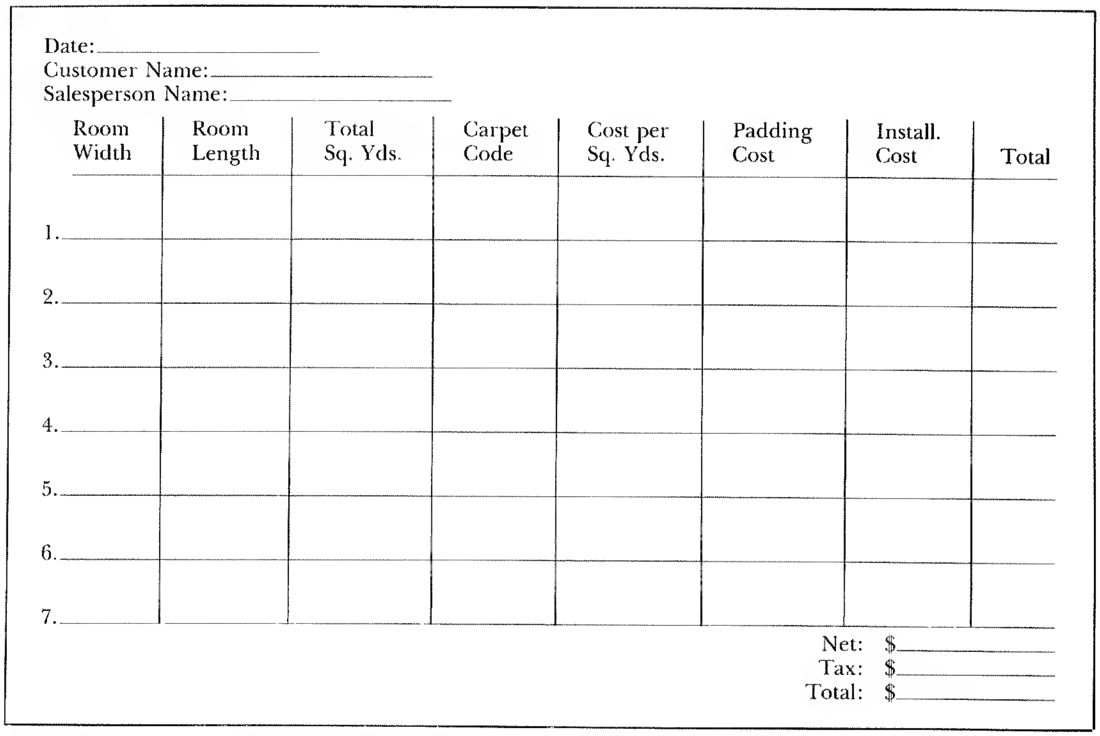
The second problem is that the store has enjoyed a tremendous growth lately, and the salespeople have many customers waiting for estimates. Can a computer system help make the estimation procedure quicker so that no additional staff have to be hired?

You start your analysis by asking your friend to describe the procedure a salesperson currently uses to provide a customer with an estimate. He gives the following explanation:

"The customer gives the salesperson the dimensions of the room to be carpeted. The length and width are given in feet. The salesperson writes these dimensions down on a customer estimate form. Since we only sell carpet by the even yard, these dimensions must be rounded up to the next even foot that is divisible by 3. For example, a room that is 10 feet, 6 inches long by 13 feet, 10 inches wide is converted to 12 feet by 15 feet.

"Next, these dimensions are multiplied to give the area of the room in square feet. This number is then divided by 9 to give the area of the room in square yards. This area is also written on the form. The customer then tells the salesperson the cost of the carpet the customer would like to install. This cost per square yard is multiplied by the area of the room in square yards to arrive at the total cost of the selected carpet for that room. This final result is written on the form, which is then given to the customer."

You then request a copy of the customer estimate form. This is shown in Figure 1.2-1. Note that it allows for estimates to be given for several rooms on the same sheet, in case the same customer has more than one room to carpet.



Date:

Customer Name:

Salesperson Name:

	Room Width	Room Length	Total Sq. Yds.	Carpet Code	Cost per Sq. Yds.	Padding Cost	Install. Cost	Total
1.								
2.								
3.								
4.								
5.								
6.								
7.								

Net: $

Tax: $

Total: $

FIGURE 1.2-1: A customer estimate form for a carpet store.

The main function of the program your friend needs is to generate this form for the customer. The general description of the system is now fairly easy to create:

This system will generate an estimate for carpeting up to seven rooms. The program will be operated either by a salesperson or by the customer. The customer will provide the dimensions of all rooms to be carpeted, along with the per-square-yard cost of the carpet selected for each room. The program will then generate a report that summarizes the total cost of carpet for each room, and the total cost of carpet for all rooms combined.

This description calls for a system that will exactly mimic the function that has been performed by a salesperson in providing an estimate. However, it is perfectly reasonable to assume that this procedure could be enhanced to provide additional services to the user, or to take other factors into account when producing the customer's estimate. An example might be adding in the cost of padding and/or installation, if the customer wants these.

Example 2: A Carpet Store System

Some time later, you run into your friend with the carpet store and you ask him how the new computerized estimate system is working. He says that it has been a great success, and he would like to computerize more of his operation. So you must again analyze his current operations, including his record-keeping setup. From further questioning, you learn the following:

The computerized customer estimate is used as a basis for filling out an official salesperson estimate form (shown in Figure 1.2-2). This form gives additional details, and is later used to place an actual carpet order. It also includes the cost of padding and/or installation, as well as discounts the customer is entitled to.

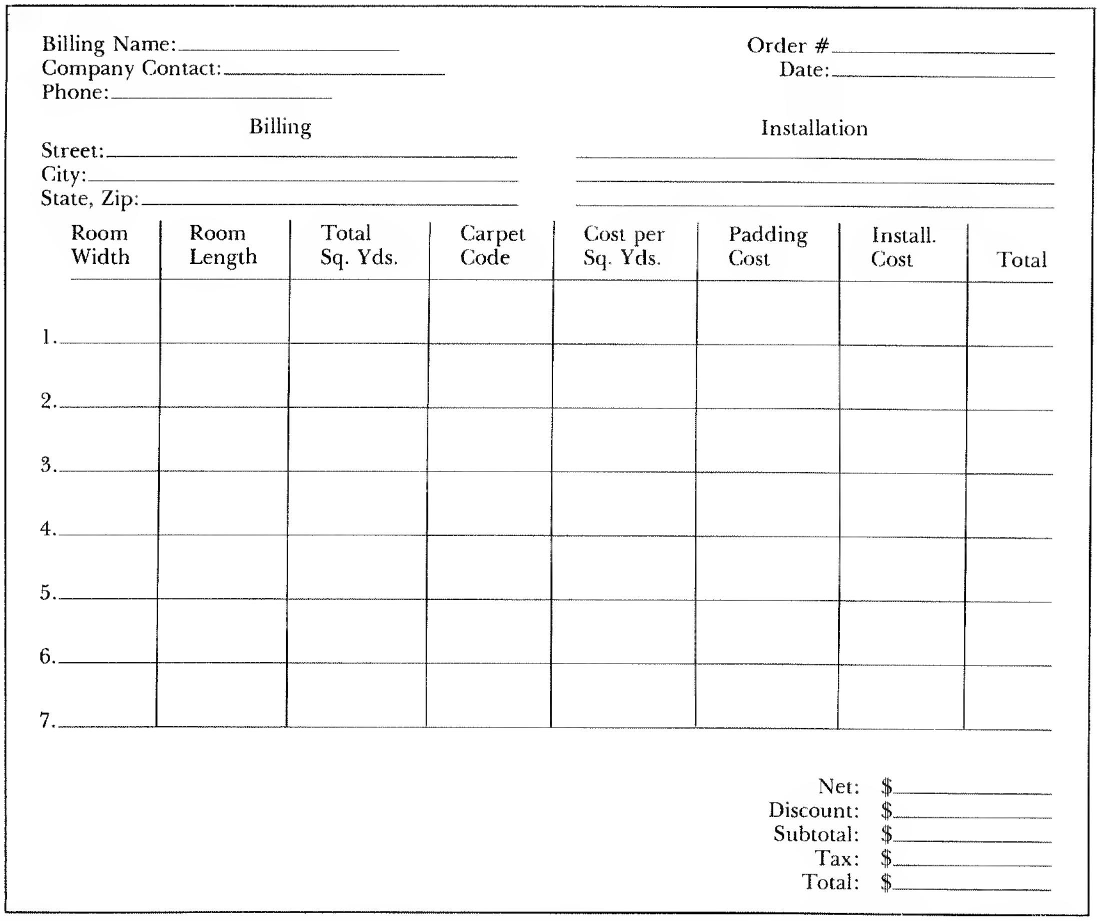
Discounts are calculated based upon a formula that takes into account certain factors, such as the customer's purchasing history over the last year. The following discount is applied based on the amount of the current order: for an order less than \$2000, no discount; for an order greater than or equal to \$2000, but less than \$5000, a 5% discount is given; for an order greater than or equal to \$5000, but less than \$10,000, a 7% discount; for an order greater than or equal to \$10,000, a 10% discount. In addition, a regular customer (any customer who has purchased more than \$7500 worth of goods in the last 365 days) receives an extra 5% discount. Credit orders are not eligible for a discount, except for regular customers.

This salesperson estimate is kept in a current estimates file for thirty days, during which the customer can order the items on the estimate at the guaranteed price given. If the customer does not order anything from that estimate within thirty days, the estimate is removed from the file and discarded. If the customer does wish to order any item(s) from the estimate, the estimate is removed from the estimate file and additional information, such as the installation address, is taken. Finally, the total is recomputed. This is done because it could affect the amount of the discount the customer may receive. In addition, the current per-square-yard cost of a carpet is used whenever this is less than the cost listed on the estimate, such as when the carpet goes on sale after the estimate was made. In other words, the customer is always guaranteed the lowest price on a carpet.

Finally, the estimate form (now called the order form) is placed in the order file. This order form will be used for scheduling delivery and installation.

All orders are net thirty days unless credit has been approved. In addition, a regular customer's order is immediately scheduled for delivery and installation. A non-regular customer must wait until credit has been approved before delivery and installation are scheduled.

The above provides the necessary information to create a general



Billing Name:______ Order #______
Company Contact:______ Date:______
Phone:______

	Billing	Installation
Street:		
City:		
State, Zip:		

	Room Width	Room Length	Total Sq. Yds.	Carpet Code	Cost per Sq. Yds.	Padding Cost	Install. Cost	Total
1.								
2.								
3.								
4.								
5.								
6.								
7.								

Net:	$
Discount:	$
Subtotal:	$
Tax:	$
Total:	$

FIGURE 1.2-2: The official salesperson estimate form, which includes additional information about the customer and calculates discounts.

description for a system that will computerize the carpet store order system. There are some human dynamics that must be taken into account, however, before a final description will be useful in the preliminary design phase. In addition, there are certain pitfalls that must be avoided when converting this system to a computer.

First, who is doing what in the system described above? It appears that the customer is responsible for generating a preliminary estimate using the previously created customer estimate program. This is then taken to a salesperson who will create the salesperson estimate. The form generated from this action is next given to someone (maybe a secretary) who adds it to the estimate file. When a customer later wants to place an order, the secretary must retrieve the estimate form from the estimate file and return it to the salesperson who completes the necessary information to generate the order form. This order form is given to the secretary who places it in the order file. The orders in the file are then used by the delivery and installation crews to perform their functions. Finally, it is the secretary's job to purge the estimate file of estimates older than thirty days.

A first approach to designing a new system might be simply to convert the manual system into one that uses a computer to keep track of the various forms and to perform any necessary calculations. Thus, the estimate file and the order file could be envisioned as separate computer files stored on disk, instead of file folders in a filing cabinet. Additional files might be required to store other information, but this can be considered later.

It is certainly easy to create a system that will allow a user to enter the information needed for the estimate and order forms. This is already done to a certain extent in the original customer estimate program. An entry, verification, and report output program is fairly simple to create. Such a program could easily handle the calculations needed, given access to information such as a discount schedule or a cost table for different carpets.

In addition a computerized system would certainly eliminate the difficult aspects of this procedure, such as the calculations. It could also save the cost and difficulties of dealing with the special forms that are used. For instance, if an order form is returned to the estimate file instead of placed into the order file where it belongs, this will cause an error (namely that the carpet won't be delivered because the delivery person can't find the order form). Finally, such a system could eliminate the functions performed by the secretary, giving the secretary more time to do other things.

However, such a system can also create problems if not designed properly. In this case, consider how each of the functions is performed and by whom. Imagine a single computer with appropriate software to perform the functions outlined above. Is it a good idea for the customers to use the same machine as the salespeople in preparing estimates? How will the machine know which function it is to perform at any time? What happens when more than one salesperson needs to prepare an estimate at the same time? How do the delivery and installation crews access the order file for the information they need?

Some of these questions are easier to answer than others. For instance, when an order is taken, the customer must be given some record of the transaction. If a copy of the order form is printed, this can be not only used as a receipt, but a second copy can be used to inform the delivery and installation crews of the order.

In terms of the customer's unofficial estimate and the salesperson's official estimate, it does not make sense for the customers and salespeople to be competing for the same system. Since there is only a minimal connection between the customer's estimate and the salesperson's estimate (the room dimensions and the cost of the carpet), this information can easily be reentered by the salesperson when creating the official estimate. As a result, these two estimates can be separated entirely from one another and can be placed in different machines. Perhaps a machine can be placed in the middle of the store for use only by customers to get an informal estimate. This machine would run only the program that has already been implemented for the store (as defined in Example 1, page 21).

The salespeople present a different problem. The length of time that a customer will have to wait for an official estimate depends on two things: the time it takes to input the necessary information and generate the form, and the number of other salespeople waiting ahead of his or her salesperson to prepare an official estimate. In a situation where there are only two salespeople and it takes only three minutes to complete the estimate, a single system for use by both salespeople is perhaps a reasonable approach unless there are always many customers waiting for service. But if it takes significantly longer to prepare an estimate, or if there are many more salespeople, then another method must be used or the customers will find the wait intolerable. This could cost the store customers in the long run.

A simplistic approach would be to give each salesperson his or her own computer with the exact same software. Then the customer would have to wait only if more than one other customer was already waiting to get an estimate.

This certainly solves the problem for creating the customer estimate. However, remember that one of the actions that must be taken by the system is to add any estimate to the estimate file. In the case of using two machines for the salespeople, there will be two estimate files, one in each machine. When a customer later comes back to actually place an order for which he or she previously received an estimate, which file do you look in?

This could be resolved by using the salesperson's name to help identify which file the estimate is in. In fact, it makes some sense that the salesperson who took the estimate should also take the order, especially if he or she is on a commission. But this scheme goes somewhat awry if the original salesperson is out sick or on vacation, or is busy with another customer when the customer returns to place the order. In addition, what do you do if the customer has lost his or her copy of the original estimate?

Another problem of the system described above is that it would make adding functions to the system extremely difficult. For instance, what if the store owner later wants to add a facility to generate a mailing list from the estimate and order files? Having multiple files for the same function would make this very difficult.

Finally, the mechanism for calculating discounts requires keeping particular data about each customer. This data must be updated every time a customer makes a purchase. This function would be greatly complicated if each of the computers needed its own copy of this customer information file. With multiple copies, the accuracy of the information (called the **integrity** of the file) is much more difficult to ensure.

Although a system can be created that would, for instance, update the main files for estimates, orders, and customers at the end of the working day, such a system is fraught with danger and inconvenience. A simpler solution would be to return to the single computer idea. However, instead of having a single access point to the system (e.g., a single keyboard and monitor), a multi-terminal system could be used. In this way, each salesperson could have a terminal on his or her desk hooked to the main system. Salespeople could all access the same file "simultaneously" (i.e., close enough to count), so that only a single copy of the files would be needed. Such a **multi-user system** could also provide access to the information contained in the files to other employees, such as the secretary and delivery crew.

A third approach would be to use a **network** system, which connects multiple computers so that they can access common data. Such a system typically provides storage on a hard disk of up to 100 megabytes and access to a shared printer. This can be more cost effective than a multi-user system if the store already has more than one computer.

A small store would probably begin with a single computer located on a salesperson's desk. As the store grew, however, it might add a second computer, suffering with the inconvenience of multiple copies of the data for a while. Finally, as the store became even more successful, it might add a network system or convert to a multi-user system.

1.3 MODULAR DESIGN

Once the general description of the program is completed, we should have a good feel for some of the program's functions. However, the program's description is still so general that we should not jump directly into the design phase. Instead, we should perform a preliminary design, which includes defining the functions of a program in more detail, defining some of the data that will be used, and simplifying some of the logic that will be part of the program.

Our ultimate goal, naturally, is to reach the production phase, the point in the life cycle where the system will be used to perform real work. Maintenance is essentially concurrent with production because of the frequent need for making changes to the system. Such changes may be due to errors that were not discovered during testing, modifications to the physical environment (e.g., a new computer was purchased), or modifications to the system's specifications.

Now that we have a general idea of the application's overall function, it is time to begin breaking this rather broad description into smaller pieces to get a better feel for the exact functions that must be performed. As C.A.R. Hoare, one of the most respected computer scientists, put it, "Inside every large problem is a small problem struggling to get out." While this is perhaps more an indication that most large problems are really only small problems in disguise, it can also be taken as an indication that every large problem is best attacked by breaking it down into a number of smaller problems that, individually, can be easily resolved. This concept will be referred to as the **principle of modularity.**

The rest of this chapter is concerned with turning the general description (the large problem) into more detailed descriptions of the various functions (the smaller problems) that the application must perform. This results in a preliminary design that serves as a base for the design of the program's logic. In addition, it provides an opportunity to evaluate the application by providing some detail about the application's functions, without it being necessary to fully implement the functions.

Modules

Up to this point, we have been most concerned with the functions that a program will perform. These functions usually provide natural dividing lines within the program, and will form the basis for implementing the code. However, up to now we have been treating these functions informally.

The term **module** can be used more formally to describe a function. A module is a section of the program that performs exactly one function. This section might be a **block** of code (a few lines of code that are related) or an entire subroutine. The main concern is that each module is self-contained. If a particular module is removed from the system, only the function that that module performs should be affected.

The main task to be accomplished in the preliminary design is to define the various modules of an application. This is done by treating each function of the application as if it were a separate program, and creating a general description for each function. Whether each function will then be a small block of code or a complex subroutine is immaterial at this point.

It is not sufficient, however, simply to call each function so defined a "module." The requirement that each module have only one purpose is not always easily met, but is essential to the concept of modularity. By observing some of the characteristics that "good" and "bad" modules exhibit, we can begin to develop commonsense approaches to dividing up a large task into smaller ones.

Module Cohesion

First, the term **cohesion** can be used to indicate the uni-functionality of the module, i.e., whether the module does indeed adhere to our single function requirement. A "good" module has strong cohesion; all statements within the module are strongly related to one another. If the statements are strongly related, then they collectively perform one function. If a module were developed to perform more than one function, then the statements of two functions would not be strongly related, and the cohesion of the module would be described as weak. An exceptionally weak module, the opposite of a cohesive module, would be a collection of unrelated statements.

The object, then, is to define cohesive modules. Some modules of a particular program may be more cohesive than others, since the dividing lines of functions are sometimes fuzzy at best. This is not necessarily worrisome. However, strive for defining modules that exhibit as much cohesion as possible, in order to ensure that the principle of modularity is exploited to the fullest.

Module Coupling

A second area of concern is a module's relationship with other modules. The notion of **coupling** describes those elements of a module that are shared with other modules. Such sharing makes one module dependent on other modules to some extent. It is the extent to which these modules are coupled that is of concern.

The object is to create modules that are as independent of one another as possible, so that a change in one module will not affect the functioning of any other module in the system. This isolates changes to as few modules as possible. In addition to saving time when changes are made, this improves the quality of the program by limiting the number of modules that can be infected by the introduction of a bug.

The obvious theoretical limit of coupling is for all modules in a system to be completely independent, with no coupling present. However, it is usually impossible to reach this limit in practice. The problem is that modules can be related to one another in many different ways. Elimination of one kind of coupling may increase another kind. In addition, modules in a system are naturally related by the simple fact that all the modules perform functions related to the goals of the application.

The most common form of coupling is **data coupling,** where data or data descriptions (e.g., data definitions in Pascal) are shared between modules. When such sharing is handled using a highly formal protocol, the resulting data coupling is usually of little concern. Appropriate mechanisms for sharing data between modules are discussed in Chapter 3 (3.2 Implementation Guidelines). Such coupling is generally necessary in order to avoid the extremely difficult and time-consuming procedures that would be necessary to totally eliminate data coupling, and so is well tolerated.

Top-Down Development

Now that we have some notion that large problems can be broken down into smaller problems that are (we hope) easier to resolve, *how* do we go about this modularization of a program? It is quite important that a program be divided along natural lines, and that modules are selected so as not to make the definition of later modules difficult or unnatural.

Learning to identify natural modules in a system comes only with practice. At first, defining such modules for a system may take several attempts. It is not uncommon even for professionals to find that the design they have been developing won't be satisfactory because of the division of the modules selected.

Most professionals use a method called **stepwise refinement** when developing a design for a system. A more informal name for this technique is **top-down** development. The formal name indicates that the method will be iterative, i.e., is a set of steps that are followed multiple times, with each iteration giving additional refinement to the design. The informal name indicates the starting point of the method, at the topmost level of the design.

The top of our design so far is the general description of the application. We will begin with this description of the large problem, and successively add levels of more detailed description below it. Each level is another step of refinement. We continue breaking the levels down and defining additional levels until adding more detail would require that we write code. The process stops when all functions that have been identified have been defined to this level.

For example, consider again the description of a checkbook system that was given in the previous section:

The program will help the user to maintain his or her checkbook and to balance the checkbook automatically at the end of each month. The program will keep track of all checks and deposits of the account. With this information, each month's bank statement can be balanced simply by entering the final daily balance listed on the statement. The program will generate a new balance, which is then compared to the final balance in the user's check register. If these values are the same, then the checkbook is balanced. If the values are not the same, then the user has made a mistake in entering information about the checks or deposits.

Now begin to break this general description down into smaller pieces. Each piece should describe a particular function that the program must perform. The collection of all of the functions so described should be the definition of the program.

This program follows a format that can be applied to most programs: input, process, and output. First, the program must perform input operations in order to collect enough data to do something. Next, the program processes the data that it just collected. Finally, the program outputs the results.

Several types of input are required for the program. First, the program is keeping track of all the checks and deposits ever made to a particular account. This implies that a file that mimics the check register must be available. So, as one input, the program must have some mechanism for reading this file into memory.

Second, the user must be able to enter new checks and deposits into the file. This is a separate function from the one just described, since it implies not only inputs from the user but also inputs and outputs to the file.

Third, the user must input certain other information, such as the final daily balance listed on the monthly statement. This again is a separate function.

What processing must be done in the system? The program must do some computations in order to generate a new balance. These computations include calculating the total for all outstanding checks, the total for all outstanding deposits, and the new balance. The computation of the new balance can be thought of as a single function that this program must perform.

In analyzing the computations, however, we have discovered additional functions. How does the program go about totalling the outstanding checks and deposits? We could think of there being a function that totals outstanding checks, and another that totals outstanding deposits. This is appropriate since these actions will undoubtedly require accessing the file that contains all the checks and deposits.

What about the output function? The result of the program is supposed to be an indication of whether or not the checkbook is balanced. Although this final determination will probably be done by hand, the program should at least output some type of report that gives, for instance, the total amount of outstanding checks, the total amount of outstanding deposits, the old balance, and the new balance that will be used to determine whether the checkbook is balanced. A more complete report might include lists of the outstanding checks and deposits.

This output could be viewed as a single function, since it, in effect, generates a single report. However, there might be separate functions for generating the lists of outstanding checks and deposits and the report itself.

Is this all of the functions that need to be defined for this simple program? What about the statement that if the checkbook is not balanced, then the user has entered something incorrectly? If there is an error in the file containing the checks and deposits, how is this error going to be corrected?

There must be another function which allows for entries in the file to be changed, just as it must be possible to change entries made into the check register itself. This function might be viewed as overhead that is often required when dealing with files. However, it, in combination with another previously defined function, might be thought of as a function that permits modifications to the check register file. Such changes might be to add new checks and deposits to the register (the previously defined function), to delete entire entries, or to make changes to entries.

This last part could itself be broken down into sub-functions. First, changes could be made to an entry because an error was made when the entry was originally input. Second, a change could be an update of the status of an entry, as in the case when a check's status changes from outstanding to cashed.

Getting the Big Picture

The example above shows how the general description can be used to help define the various functions of a program. In addition, it shows how these functions can be further broken down into components that are sub-functions. However, the verbal description of this process is somewhat difficult to follow. Even though this technique yields important results, it would be worthless if these results make it no easier to design the program's components.

Our ideas of top-down presentations and modularity can help us begin to divide a task into a series of sub-tasks, each sub-task performing a particular function. By then creating a hierarchy of all functions, we can better see how the various functions interrelate. In addition, we can easily "count noses" to be certain that all the functions that the particular system is supposed to perform are, in fact, represented.

By using a graphical representation of this hierarchy, we can more easily grasp the overall picture of how the various pieces fit together. This is really not a new idea. Figure 1.3-1 shows such a hierarchy chart for a university administration. It is used as an organizational chart so that lines of command can be easily defined and recognized. While fairly standard and easily readable, note that such an organizational chart nevertheless can contain ambiguities. For instance, is the Assistant to the President at the same logical level as the Vice President for Academic Affairs? That depends on the interpretation of what the solid lines mean. If they are interpreted to indicate a level of responsibility, then the answer is probably no. However, if they are instead interpreted to simply indicate who reports to whom, then the answer is that the president's assistant is at

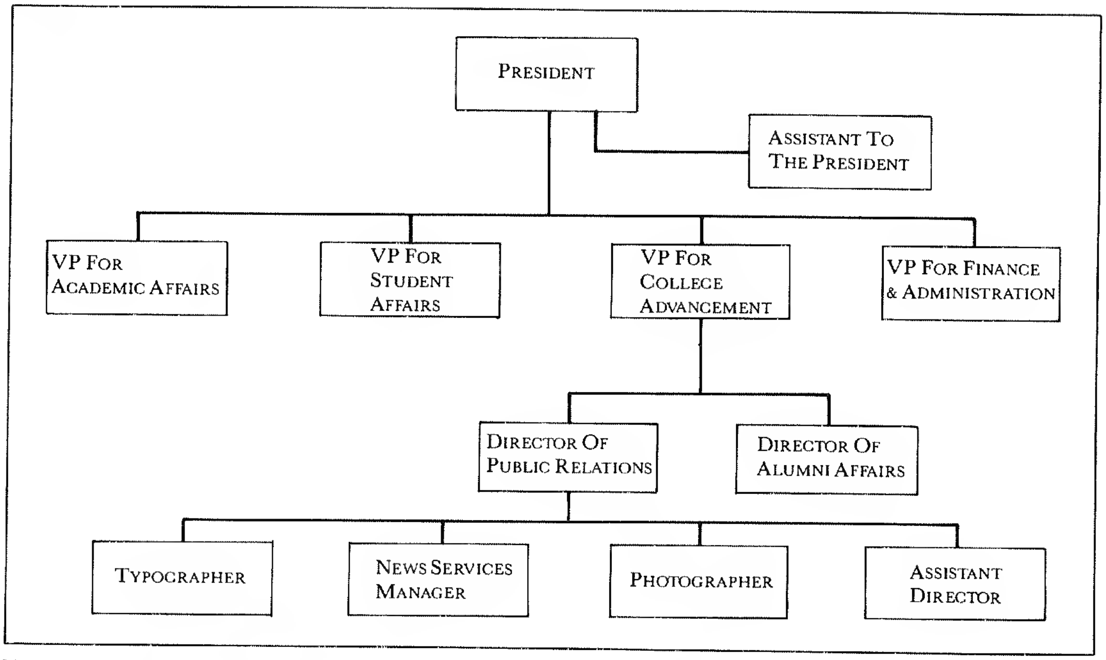

PRESIDENT
ASSISTANT TO THE PRESIDENT
VP FOR ACADEMIC AFFAIRS
VP FOR STUDENT AFFAIRS
VP FOR COLLEGE ADVANCEMENT
VP FOR FINANCE & ADMINISTRATION
DIRECTOR OF PUBLIC RELATIONS
DIRECTOR OF ALUMNI AFFAIRS
TYPOGRAPHER
NEWS SERVICES MANAGER
PHOTOGRAPHER
ASSISTANT DIRECTOR

FIGURE 1.3-1: A hierarchy chart for a university administration.

the same level as a vice president. Actually, the Assistant to the President's box was placed on a line separate from the vice presidents' line to help avoid the former interpretation.

Building a hierarchical chart is generally an iterative process. This allows us to define categories or functions on a very high level, and then successively refine subcategories or sub-functions until a satisfactory level of detail is achieved.

HIPO Charts

Several years ago, IBM formalized a method of constructing hierarchy charts and defining details about the functions in the hierarchy. This method, known as the HIPO (Hierarchical plus Input-Process-Output) technique, is really two methods that help to accomplish the final goal. The first is a representation of the overview, and is called the H-chart or Visual Table of Contents (VTOC). The VTOC is a formalized method for constructing organization charts for program systems. The second method is the IPO chart, which provides details about the inputs, processes, and outputs of an individual block of the VTOC.

HIPO charts can be used for a variety of purposes in a system development project. First, HIPO charts are used retroactively to document existing systems. While this is certainly not the most effective use of HIPOs—since they are not used to aid in the design of the system—it can help when a system later needs to be modified. Any documentation is better than nothing, and the HIPO charts provide a quick, easy, standard reference for a system's functions on a high level.

Second, as will be the case in this book, HIPO charts can be used to help organize a new system's requirements and specifications. Here the technique is used to construct a system overview and a preliminary design of the system's components.

Third, HIPO charts are used by some to carry the design to a sufficiently detailed level so that implementation can begin. Personally, I find this particular use of HIPOs to be unwieldy, in that HIPO charts have a tendency to become too lengthy. Shooman [97] tells of a room at RCA that was covered by HIPO diagrams floor to ceiling, with the number of diagrams well into the hundreds, all for a single military system. Such devotion to a tecchnique can ultimately become more trouble than it is worth.

Finally, HIPO diagrams are often used during the "installation" stage of a software package to support the installation and the training of personnel who will use or supply information to the new application.

There are many advantages to using a technique like HIPO charts. One is that nearly anyone can understand the charts; you don't have to be a computer professional with fourteen years of experience to know what they mean. This is partly because they use a graphical form that is already familiar to a majority of the people involved, and partly because much of the detail of the IPO chart is written in English. The systems analyst, the programming staff, management, and the application user can all look at the same preliminary design document and understand the functions involved. This makes it much more likely that the software resulting from the project will, in fact, satisfy the needs of the user. Therefore, the HIPO method provides for a common communication medium for the entire staff involved in the project.

In addition, HIPO diagrams can easily be constructed as a group effort. Since most sizeable projects require that numerous people be involved, at least in the initial design phases, the diagrams are an effective medium for involving personnel on a variety of levels.

Another advantage is that the HIPO method naturally supports the top-down and modular techniques already discussed. Effectively, then, the HIPO method is the instrument used to bring modularity and the top-down theory into practice.

Finally, using HIPO diagrams during the preliminary design phase allows the designer tremendous freedom in experimenting with the design. This is possible because the details of the system are not yet considered. In addition, HIPO diagrams are very easy to change, since they are involved with *what* the system is supposed to do, not *how* the functions are to be accomplished.

The Visual Table of Contents (VTOC)

As the name implies, the VTOC, or H-chart, will be used to graphically represent a table of contents for the application, outlining all the necessary functions for the system. The organizational charts in Figures 1.3-1 and 1.3-2, while not H-charts in the strict sense, are somewhat typical of what is used in designing program systems.

Note the use of a hierarchical numbering system in the boxes of Figure 1.3-2. This provides a handy reference to identify the sub-functions. In addition, the numbering system indicates the exact level of the function being discussed, as well as to which higher-level function the particular sub-function belongs. These reference numbers can be used to construct a type of legend to accompany the chart, which gives more details about a particular box.

Recall our carpet store example from previous sections. Let's now design a VTOC for this application. The VTOC will later be used for:

1. presenting an outline of the application as we see it to the user, in this case the carpet store owner;
2. defining modules;
3. forming the basis of the design of individual functions.

Figure 1.3-2(a) shows the highest level functions for this application. During the interview with the user we discovered that there were several things the owner wanted from a system, namely a setup that would allow customers to calculate their own estimates if they wished (box 2.0), a more accurate estimate prepared by a salesperson that would include any discounts the customer might be entitled to (3.0), an order entry system (4.0), and a way to generate mailings to customers (5.0). These have been divided into separate functions, even though, as we shall see, some of the sub-functions will be similar or even identical.

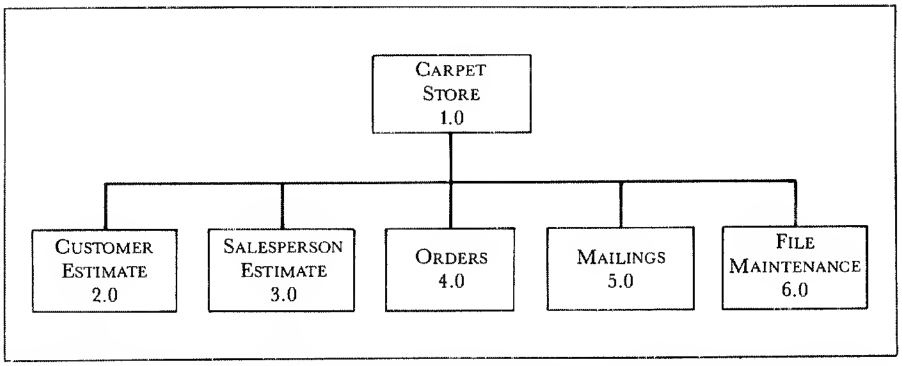
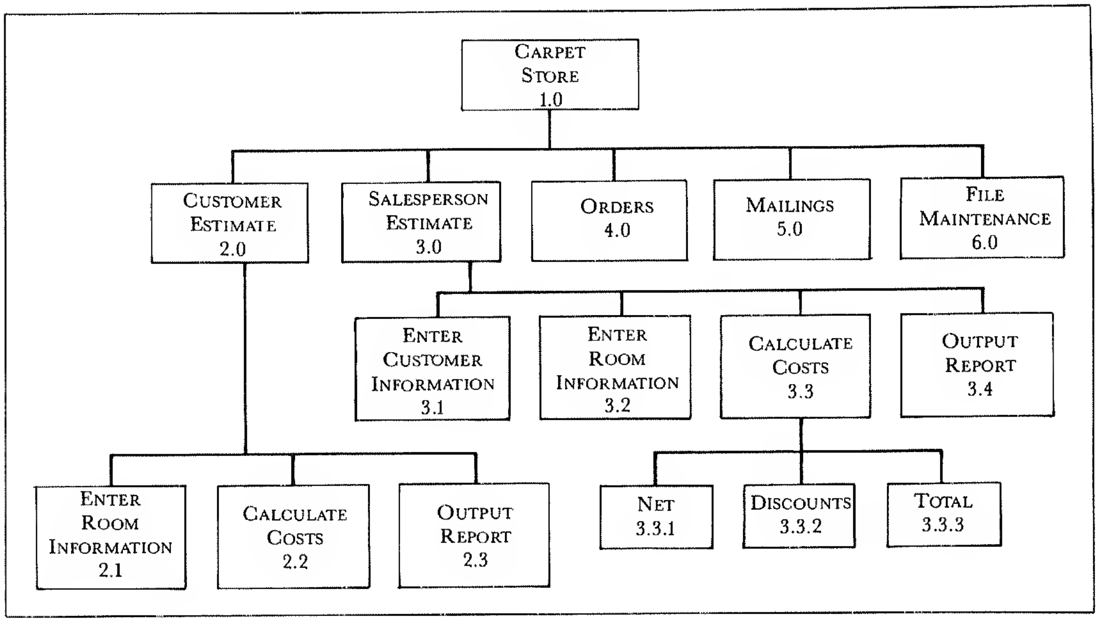


FIGURE 1.3-2a: **These organizational charts, while not H-charts in a strict sense, are somewhat typical of what is used in designing program systems. They show the highest level functions for a carpet store application (a); sub-functions of the "Customer Estimate" function and the "Salesperson Estimate" function (b); and the "Orders" function (c).**

Note that one additional function has been added that the user didn't specifically ask for, namely file maintenance (6.0). This is one of those functions that is common to most sophisticated systems. The general purpose of such a facility is to help the user to correct any mistakes that might have been entered into a file. Without some mechanism to modify the files outside the regular system functions,

FIGURE 1.3-2b



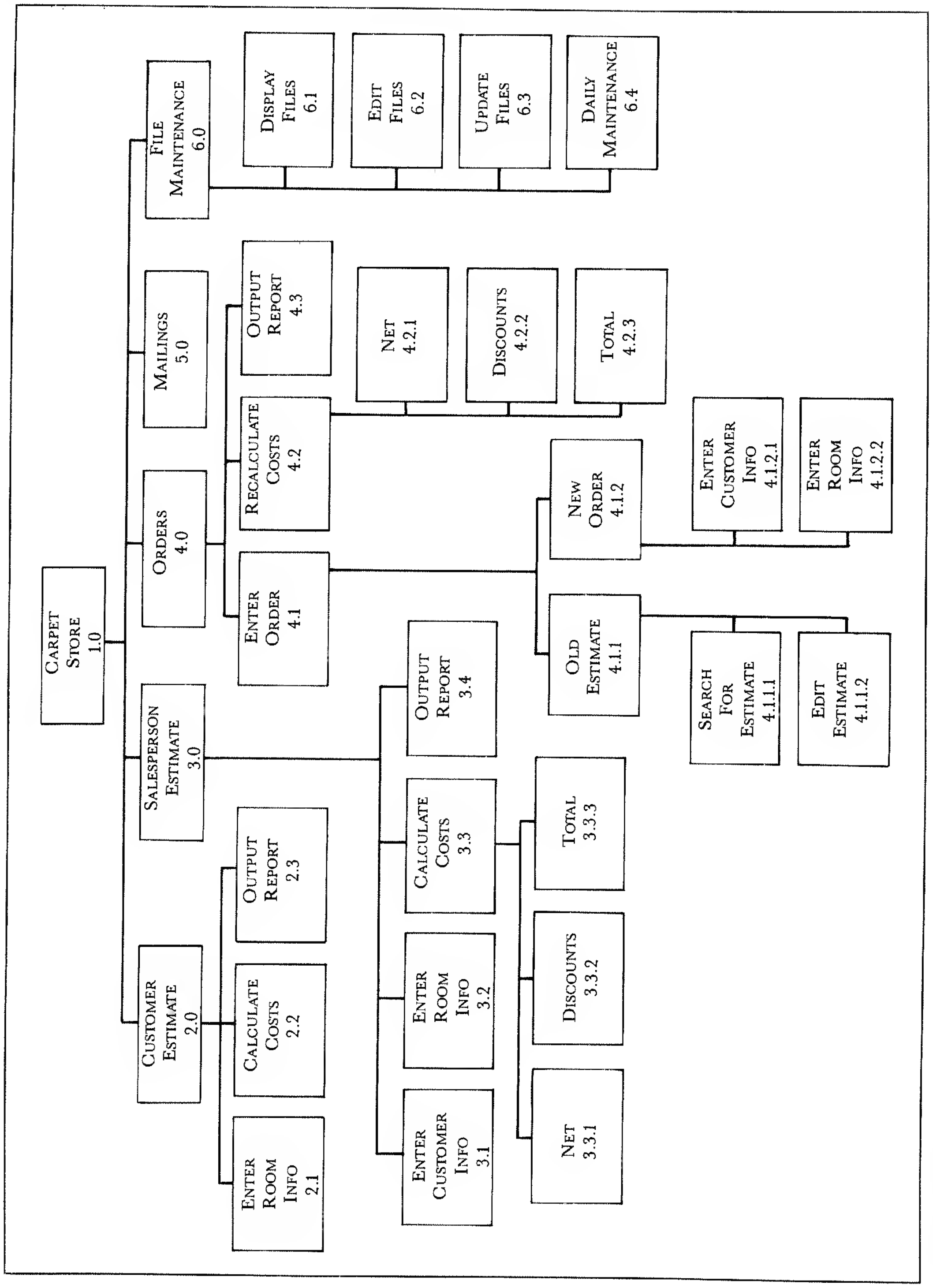

CARPET STORE 1.0
CUSTOMER ESTIMATE 2.0
ENTER ROOM INFO 2.1
CALCULATE COSTS 2.2
OUTPUT REPORT 2.3
SALESPERSON ESTIMATE 3.0
ENTER CUSTOMER INFO 3.1
ENTER ROOM INFO 3.2
CALCULATE COSTS 3.3
NET 3.3.1
DISCOUNTS 3.3.2
TOTAL 3.3.3
OUTPUT REPORT 3.4
ORDERS 4.0
ENTER ORDER 4.1
OLD ESTIMATE 4.1.1
SEARCH FOR ESTIMATE 4.1.1.1
EDIT ESTIMATE 4.1.1.2
NEW ORDER 4.1.2
ENTER CUSTOMER INFO 4.1.2.1
ENTER ROOM INFO 4.1.2.2
RECALCULATE COSTS 4.2
NET 4.2.1
DISCOUNTS 4.2.2
TOTAL 4.2.3
OUTPUT REPORT 4.3
MAILINGS 5.0
FILE MAINTENANCE 6.0
DISPLAY FILES 6.1
EDIT FILES 6.2
UPDATE FILES 6.3
DAILY MAINTENANCE 6.4

FIGURE 1.3-2c

such correction becomes very tedious. We'll examine the details of the file maintenance function later.

The massive carpet store problem has now been split into five smaller, more manageable problems. Note that the inclusion of these functions in this chart does not imply anything about exactly who the user of each function is. Indeed, the user of each function may be different. Certainly this is implied by there being two ways to get an estimate for carpeting, either by the customer doing it for himself or herself, or by the salesperson doing it. In addition, the store owner may not want salespeople spending their time on clerical duties such as preparing mailings to customers or file maintenance, so there might be a different user or users for these functions.

The chart also is not meant to imply any particular order to the execution of the functions defined. The order in which these things occur will be random to a certain extent. Some of the activities could be simultaneous, such as the customer estimate and salesperson estimate functions. One customer could be calculating an informal estimate while the salesperson is busy calculating a formal estimate for another customer. More important, these functions could be executing on entirely different machines, with one set up on the display floor just for customer estimates, the other sitting on the salesperson's desk.

The legend associated with this chart briefly describes each major function. The descriptions are strictly informal but will later be expanded in other documents to include many more details. At this early stage of design, keeping everything simple is the key to the success of this technique.

Figure 1.3-2(b) shows the sub-functions that might be added under the "Customer Estimate" and "Salesperson Estimate" functions (2.0 and 3.0). While these two functions perform similar tasks, the main difference is that the customer estimate is not official, since it does not take any possible discounts into account when calculating the total. Therefore, the "Calculate Costs" sub-function of the customer estimate (2.2) is not as elaborate as the one for the salesperson function (3.3). Also, note the sub-function for entering information about the customer in the "Salesperson Estimate" function (3.1). This information isn't needed in the informal customer estimate.

Figure 1.3-2(c) includes the details of the "Orders" function (4.0). Here, what was previously an estimate is changed to an actual order. Remember the store manager's description of how this is handled using physical files. An estimate sheet is found either by an order number or a customer's name in a folder that contains all estimates made within the last thirty days. This form is then transferred to another folder that contains real orders for carpeting.

There are several complications to this procedure. First, there is the possibility that a customer would like to place an order without having first had an estimate drawn up. This requires special handling in the order system. Additional information must obviously be entered, such as the customer information (4.1.2.1) and the specific room information (4.1.2.2). Second, the customer might have changed his or her mind about the particular order, and either deleted from or added to the room(s) to be carpeted, or changed the type of carpeting in a particular room. Third, the actual cost of an order might change because of changes in the cost of carpeting (perhaps it has gone on sale since the estimate was made) or in a change in the customer's discount eligibility. The idea is always to give the customer the best deal possible, so these factors might be taken into account.

Finally, note that the format of the sub-functions presented in the chart is slightly different than before. The boxes under 4.1 and 4.2 are connected by a line going down instead of across. This is merely a way of fitting several boxes into the available space and does not change the meaning of the boxes. This works only if each of the boxes (such as 4.2.1, 4.2.2, and 4.2.3) is terminal, i.e., has no further sub-functions underneath it.

Some Helpful Hints

There are several points that you should keep in mind when developing a VTOC for an application. First, as mentioned previously, there is not a single VTOC chart that could be drawn for a particular application. It is analogous to finding a path for driving to work. While there may be a finite number of ways to do so, there is not necessarily one best way, nor even a most obvious way. What works best for you may not work for someone else. Also, the path you follow on one particular day may be different than what you would follow some other day.

Second, use the VTOC for a first round of debugging your design. Make certain that all functions you feel are needed in the system are accounted for somewhere. This does not necessarily mean that there is a box with that function's name on it, but instead that you at least have it in your mind which box that particular function is a part of.

Third, don't be afraid to change your VTOC, even drastically or to the point of scrapping an initial design entirely. If a function doesn't seem to be fitting into the VTOC, it probably means that some other function was improperly defined. Remember that any mistakes in the design are much easier to fix now than later.

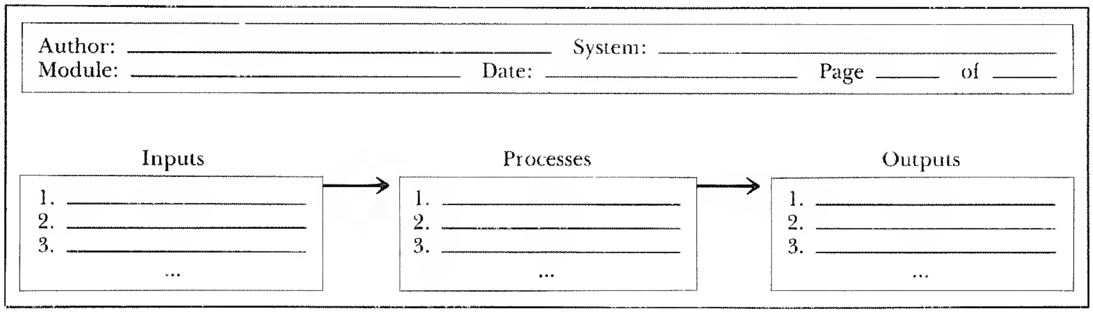
Finally, sub-functions can be defined down to as low a level as desired, to the point where coding details could be presented. However, a good rule of thumb is that as soon as you need any detail about *how* a function is going to be implemented in order to break it down further into additional sub-functions, you stop. For instance, in the "Calculate Costs" sub-function (3.3) of the "Salesperson Estimate" function, trying to break the "Net" sub-function (3.3.1) down any further would require knowing the formula for the calculation. The same thing is true for the "Discounts" (3.3.2) and "Total" (3.3.3) sub-functions.

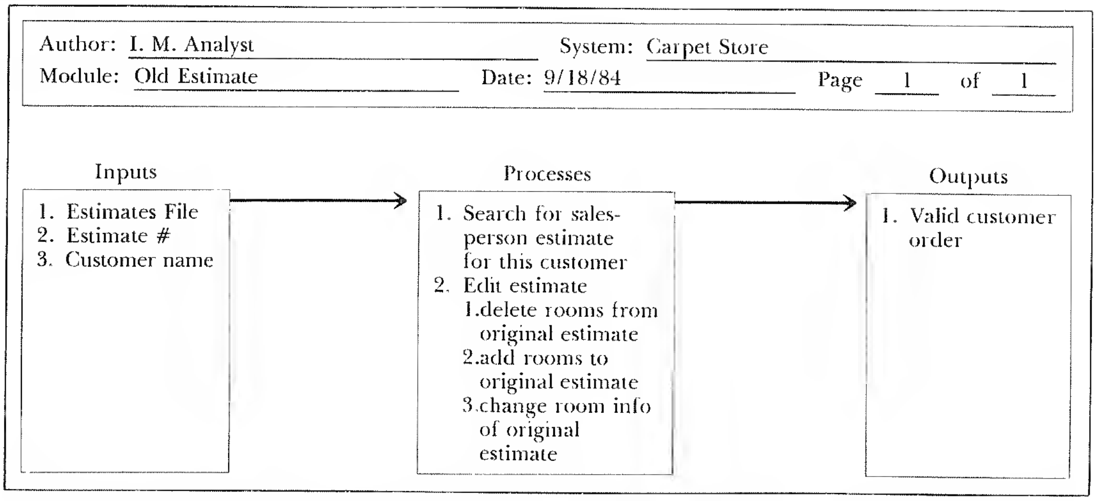
The Input-Process-Output Chart

The IPO chart is a companion to the VTOC and serves some of the same function as the legend, in that the IPO chart is a more detailed explanation of each function of the VTOC. However, unlike the VTOC, the IPO diagram includes more specific details about the functions to be performed as well as details about the inputs and outputs that will be a part of the function.

Each function is divided into three areas (see Figure 1.3-3), inputs, processes, and outputs. English is used to describe each entry in the chart, with explanations being somewhat general and avoiding details of implementation. For example, the fact that a particular

FIGURE 1.3-3: The general form of the IPO chart.





Author: I. M. Analyst System: Carpet Store
Module: Old Estimate Date: 9/18/84 Page 1 of 1

Inputs
1. Estimates File
2. Estimate #
3. Customer name

Processes
1. Search for salesperson estimate for this customer
2. Edit estimate
   1.delete rooms from original estimate
   2.add rooms to original estimate
   3.change room info of original estimate

Outputs
1. Valid customer order

FIGURE 1.3-4: An IPO chart for the "Old Estimate" function of the carpet store application.

data file is going to be implemented as a sequential file and kept in alphabetical order by customer name is immaterial at this stage. However, we must know something about the file. In this case, simply knowing the various types of data that the file contains is sufficient. This, coupled with data dictionary descriptions (1.1-4), will at least help the programmer to identify where a particular piece of data is used.

Figure 1.3-4 shows an IPO chart for the "Old Estimate" function (4.1.1) of our carpet store application. First look at the processes. Note that they coincide with sub-functions defined in the VTOC. These sub-functions could themselves be described in an IPO chart, if we thought it was necessary. Again, a useful heuristic (rule of thumb) is to stop the detailed description of sub-functions just short of when implementation issues would begin to appear. In the case of the "Edit Estimate" sub-function (4.1.1.2), a more detailed description is given in the IPO chart which describes what type of editing might be necessary. These sub-sub-functions could have been included in the VTOC, had we considered it desirable. This points out one purpose of the IPO charts. They can be used in the iterative procedure of function definitions to help us refine the overall design.

Note also the outputs of this function, namely a valid customer order. That is the object of the function.

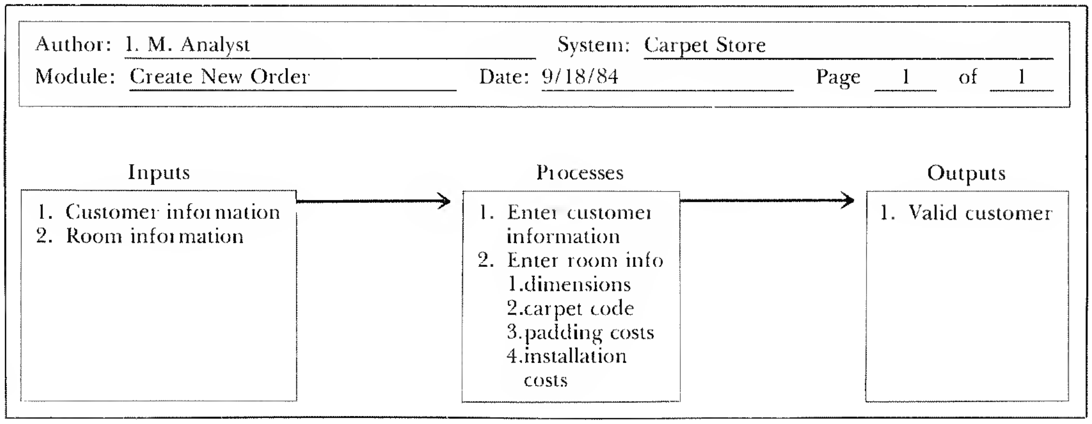
Finally, the inputs to this process are records from the estimate file created during the salesperson estimate process, and either an estimate number or the customer's name. The latter data items are used to find a particular estimate in the file. Note that at this stage we are not concerned with which data item will be used to search the file, the estimate number or the customer's name. Likewise, the processes described in the IPO chart do not include any decisions to be made. We are not attempting to describe actual program logic at this stage, only presenting a list of possible actions to be taken.

Figure 1.3-5 presents an IPO chart for the "Create New Order" function (4.1.2). Note that the output of this function is described the same way as the output of the "Old Estimate" function (4.1.1). This is because, ultimately, we do not care how a customer's order came into being, only that it indeed represents an order from a particular customer.

When preparing IPO charts, I usually find it helpful to work in a bottom-up fashion rather than top-down, probably because I generally have a better idea of the details of inputs and outputs at the lowest level first. This may seem in conflict with the top-down method used to develop the VTOC, but once the VTOC is created, it matters very little where you begin developing the IPO diagrams. Some analysts prefer to develop the same level of the VTOC and IPOs simultaneously, using the top-down approach for both diagrams. However, it is a good idea never to let the adherence to a rule or technique get in the way of practical expedience.

Another technique that is sometimes helpful is to prepare the output section of the IPO diagram first. However, since the details

FIGURE 1.3-5: An IPO chart for the "Create New Order" function of the carpet store application.


Author: I. M. Analyst System: Carpet Store
Module: Create New Order Date: 9/18/84 Page 1 of 1

Inputs
1. Customer information
2. Room information

Processes
1. Enter customer information
2. Enter room info
   1.dimensions
   2.carpet code
   3.padding costs
   4.installation costs

Outputs
1. Valid customer

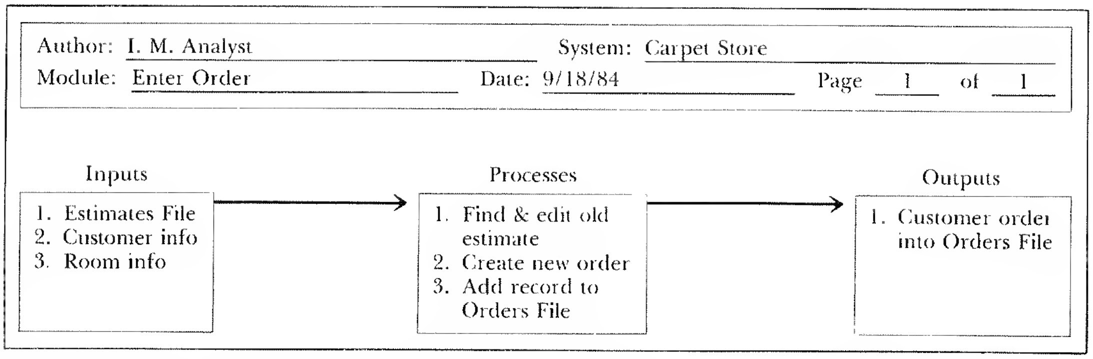


FIGURE 1.3-6: An IPO chart for the "New Orders" function of the carpet store application.

of the process section are often already known because of their being related (if not identical) to sub-functions defined in the VTOC, it might make sense to start with the process section. As should be obvious by now, the development of HIPO diagrams is at best an *ad hoc* technique.

In keeping with the bottom-up strategy, let's look at the IPO chart for the "Enter Order" function (4.1), shown in Figure 1.3-6. Its processes are clearly defined by the sub-functions already developed. Note again that, although the two processes are mutually exclusive (i.e., either the user will have an old estimate available from which to develop the customer order, or an entirely new order will have to be drawn up, but not both), there is no attempt to incorporate decision making into the IPO diagram. This detail will be left for later when the logic of a particular module will be designed.

The output of this function will be a new customer-order record for the orders file. Note that we specifically state that the record will be added to the orders file here, but didn't indicate this in the sub-functions 4.1.1 and 4.1.2. This is because it was not the job of either of these sub-functions to add this record to the order file. Adding this as a process to these two sub-functions would have meant having an identical process in each function. When this occurs within sub-functions of the *same* function, this usually indicates a redundant feature that should be moved into the parent (next higher level) function.

Finally, Figure 1.3-7 presents the IPO diagram for the "Orders" function itself (4.0). Here, details from all lower level IPOs are in-

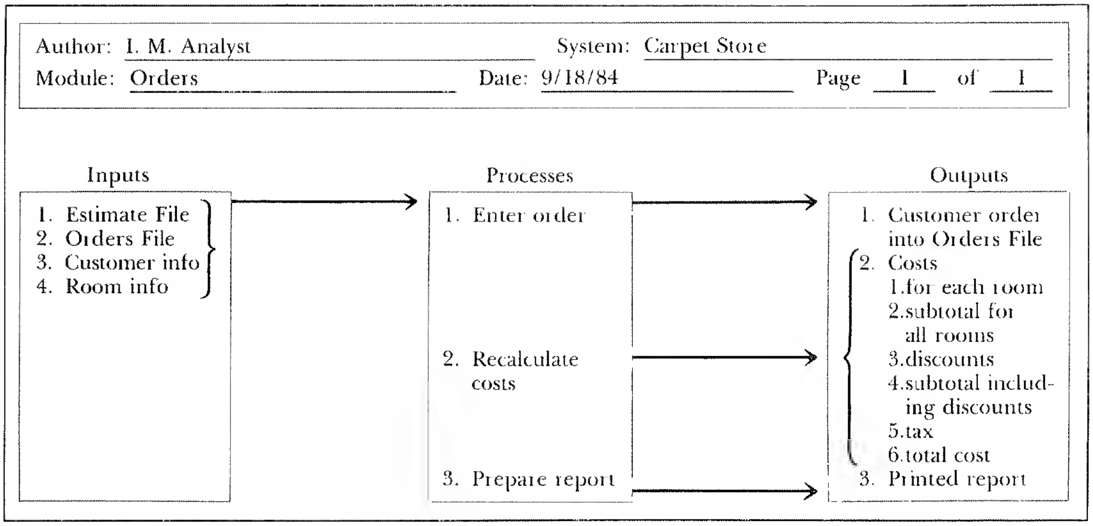


FIGURE 1.3-7: An IPO chart for the "Orders" function of the carpet store application. Note the use of the arrows and brackets to group the outputs with the processes that generate them.

corporated. Notice a slightly different technique used in the chart to indicate inputs and outputs. Multiple arrows are used to group the inputs or outputs together to indicate which process(es) they belong to. Boxes or brackets can also be used to make these groupings even more distinct. Again, the *ad hoc* characteristic of the HIPO technique allows the exact form to be molded to individual tastes.

1·4 DEALING WITH DATA

There seems to be a continuous problem during the development of a program with the "chicken or egg" syndrome. Can one really discuss what a program is supposed to do before describing the data the program will operate upon? Likewise, can one discuss data before the functions that are to be performed on that data are known to some degree? A certain amount of simultaneous development would be the ideal way of dealing with this issue. Unfortunately, we are serially-thinking beings. The best we can accomplish along these lines is to do some of the development in parallel, overlapping the development of the program description with that of the data descriptions.

This has already taken place to an extent in earlier examples. Recall that it was necessary to describe in general terms what the inputs and outputs of the various programs would be. This was the beginning of the data descriptions that will be required for each program. The data will have to be specified in much more detail, however, before the program can be implemented.

It is difficult to know exactly how much detail about data is necessary at any one point during the development of a program. In the beginning, you can be quite general, just as you can be general with the description of the program's functions. However, the "chicken or egg" problem keeps causing trouble. This is because sometimes the functions of the program are most conveniently described in terms of the data that they will operate upon. Therefore, it is important to pin down some of the details of the data fairly early.

Data descriptions are created in an iterative fashion similar to that used in developing the functional descriptions of programs. You will find at some point that the descriptions of the data you are currently using are not detailed enough to allow you to proceed further with the functional design. At such a point you will have to define the data in more detail. You might only define them in enough additional detail so that you can continue, however.

Data descriptions are especially needed early in the program development in the case of files, since many of the functions that will be defined within a program are concerned with the manipulation of files. The main files of a system will probably evolve from the manual system, such as the files that were briefly described in the carpet store example of the previous sections.

It is not possible to describe all the data that a program will use at the current stage of the development. Many data items will not be needed until the functions of the program are themselves described in greater detail. The data descriptions, therefore, will be continually updated during the development of a program, probably all the way through the implementation phase.

However, my inclination has always been to describe as much of the data as I know will be used in as much detail as I can, as soon as possible. Although such descriptions may be changed during a later phase, they at least provide a starting point. Again, the development of such descriptions can often spawn ideas for what functions are needed in the program, or for how the functions should work.

Data Attributes

Before we can discuss the main tools used to describe data, we need to be a little more specific about just what constitutes data. A data item can be thought of as consisting of certain **attributes.** The most familiar attribute is **value.** We may have only a vague notion of exactly what this term means, but most people have a commonsense feeling for its meaning. One definition of "value" might be: a numerical quantity of some type. Operations such as addition and subtraction can be performed on a "value." If, for instance, I said that the value of a car was $1300, this would be understood to mean that the car could be bought for about $1300. This number, 1300, can be used in arithmetic operations such as multiplication. In this example, the state license office might multiply this 1300 by, say, .06 in order to calculate the amount (value) of state sales tax that is owed when the car is sold.

Not all values are numeric in nature, however. We can also think of all kinds of identifiers as values. For instance, names are values when you think of a phone directory, which can be defined as a collection of all the names of people who own phones in a particular town.

This gives rise to the second attribute of data, **type.** There are three main types of data. For numeric data, there are **integers** (what we called "whole numbers" in grade school), and **floating point** numbers (any number with a decimal point in it). For nonnumeric data, the most often used type is **string.** This type can have any collection of characters as a value. Such a string might be made up of alphabetic characters, digits, or special characters such as a dollar sign or asterisk. Some programming languages have additional data types. Others, such as Pascal, even allow a programmer to define his or her own type. However, the three types listed above are the most fundamental ones, and are sufficient for defining most data.

Next, each piece of data typically has a specific **range** of possible values that are valid. For numeric types, this range is generally specified as a minimum and maximum value for the data item. There may be no theoretic limits for some data items, however. For instance, what would be the range of a data item that was to hold a value for the distance between stars? A reasonable minimum might be 0, but what of the maximum? What about the range for a data item for the gross national product? Again, 0 might be appropriate for a minimum, but what of the maximum?

In these cases, we must eventually define a *practical* range for a value. All numeric values have fixed representations which are bound by the design of the programming language and by the machine on which they operate.

For string data, there is often no real range of valid values. What is the range of valid house addresses, for instance? However, there are some string data that can have a range specified, such as a name. Can we safely describe a name as consisting of any letters, but no numbers or special characters?

Finally, a data item can have a particular **precision.** For numeric values, this is an indication of the maximum number of digits the value may have. Exceeding this maximum will result in a value that is incorrect. For instance, if the precision given to an integer data item is five, then a six-digit number would not fit. The number 123456 might be interpreted as 12345 or as 23456. Obviously neither of these values is the value we wanted.

A number's precision is a function of its type, its range, and the particular programming language and computer you are using. It can usually be thought of as a length, i.e., the maximum number of digits that the value may have. In BASIC, for instance, a typical precision for floating point numbers is given as seven digits. This does not mean that the maximum value for a floating point number is 9999999.0, however. Larger numbers can be represented by using a special format similar to scientific notation. For instance, the number 12345678.0 might be represented internally in a form equivalent to $1.234567 * 10^7$. While this is not an exact representation of the value 12345678.0, it is quite close. However, even such a small loss of accuracy can be important in many applications.

For string values, precision can be equated with size. In this case, the maximum number of characters that can make up a data item must be specified. A value any longer than the maximum will be truncated, resulting again in a loss of accuracy. The maximum length for a string is a function of the programming language you are using, but should be independent of the particular computer.

Data Definitions

A **data definition** is a formalized presentation of the form and substance of data. Its main objective is to describe a piece of data unambiguously. This description lists the various attributes for each data item. Each data item listed in the **data dictionary** is given a unique name that describes its function. Then the item is described in terms of its value, type, range, and precision, when appropriate.

The definitions are specified using a special syntax so that each definition is as clear as possible. Following are the rules for specifying data definitions.

Rule	*Meaning*
$x = a$	the data item called x is defined by the specification a, where a is itself a data item to be defined or is a complete description of the data item's attributes
$x = a + b$	the data item called x consists of both item a and item b
$x = [a \mid b]$	x consists of either item a or item b, but not both
$x = (a)$	the item specified by a is optional
$x = \{a\}$	the item specified by a can occur zero or more times
$x = y\{a\}$	the item specified by a can occur y or more times
$x = \{a\}z$	the item specified by a can occur up to z times, inclusive
$x = y\{a\}z$	the item specified by a can occur between y and z times, inclusive

Example 1: A Checkbook

In a previous section, we looked at several examples of checkbook programs, from a simple calculator to a sophisticated record-keeping system. Let's prepare some of the data definitions that might be used by the intermediate-level checkbook program that keeps track of all checks and deposits using a computerized check register.

The easiest place to start thinking about data is in connection with files. In this program, we know that we need some type of file that will hold the information that makes up the check register. So let's begin with defining what the file will look like in terms of the data that will be stored in it.

We can start with a definition such as

```
check-register-file = {[check | deposit]}.
```

This definition says that the check register is made up of zero or more items that are either checks or deposits. This is certainly how the manual check register works. This definition is complete by itself. However, it introduces two new data items, *check* and *deposit.* These new items must be further defined.

We could define the data item *check* as

```
check = check-number
        + to-whom-name
        + date-written
        + amount
        + (note)
        + outstanding-flag
```

This definition indicates that a check is made up of five or six parts, since the *note* item is optional. This definition combines data from what is written on the check with the items listed in a check register. This is desirable if we want more detailed information about the checks than that normally supplied by the check register entry alone.

Since this definition introduces still more new data items, it is necessary to continue creating definitions. We won't stop until each data item is defined in terms of format, which includes details about the attributes (value, type, range, and precision) of the data item. Such a definition can be called a **base definition.**

We could have specified the attributes for a check number directly in the *check* definition, but it is usually more understandable to use data item names when there is more than one data item in the particular definition. In this way a hierarchy of definitions is created, similar to the hierarchy used to describe program functions in HIPO charts. This makes it possible to create base definitions that are used throughout the data dictionary.

The data item names should be selected so that they are meaningful. Hyphens are used to connect words so that the result is something that resembles a name, rather than an English phrase. Meaningful names are important, since they are the only way that the meaning of the data item can be described.

It is usually easiest to continue defining the data items one at a time, from the top down, until a base definition is reached. These base definitions are the only ones that should specify the attributes of the data item. In this example, we might continue creating definitions in the following order:

```
check-number = 3{digit}5
digit = “0” . . “9”
to-whom-name = 1{character}30
character = [letter | digit | special-character]
letter = [“A” . . “Z” | “a” . . “z”]
special-character = [“!” | “@ ” | “#” | “$” | “%” |
                     "&" | "*" | "(" | ")" | "-" |
                     "+" | "=" | "," | "." | "?" |
                     ":" | ";" | "<" | ">" | "b̸"]
date-written = month
               + "/"
               + day
               + "/"
               + year
month = "01" . . "12"
day = "01" . . "31"
year = "84" . . "99"
amount = floating point number between 0 and
         99999.99, inclusive, with exactly 2 decimal
         places
note = {character}30
outstanding-flag = ["YES" | "NOb̸"]
```

There are a number of things to note about these definitions that were not discussed when the syntax of definitions was presented previously. First, quotation marks are used to indicate a character value, often known as a **literal string.** It means that the exact characters within the quotation marks are to be used. Thus, the valid values for the data item *outstanding-flag* are the words YES and NOb̸, where b̸ means a blank (which must be treated as a character like any other). This shows not only what the valid values (i.e., the range) are for this data item, but also that the type of it is string, and that the length of it is exactly three.

The two periods (. .) specify a range of values. In the *letter* definition, for example, "A" . . "Z" specifies all letters from A to Z. In this definition, note too that it is necessary to specify both upper- and lowercase letters in order to be completely general.

Next, when dealing with purely numeric data, it is usually necessary to revert to English phrases to fully describe a data item. For instance, there is no easy mechanism to define the attributes of the data item *amount.* In this definition, it was necessary to specify the data's type, range, and precision. However, a specific (**literal**) value is seldom provided for numeric items. A range should almost always be specified, however.

In this example, a check number is defined as any three- to five-digit string. This gives an implied range of "000" to "99999" for this data item, although this may not be exactly how check numbers are used in the real world. For most banks, check numbers begin with "101," for example. This could not be exactly specified using the notation of the data definition. However, it could again be handled with an English statement, such as

```
check-number = a 3- to 5-digit string, between "101"
               and "99999", inclusive
```

The *to-whom-name* has been defined as a string of between one and thirty characters, consisting of letters, digits, or special characters. This should allow any person's or company's name to be written. If thirty characters is not enough, the name will have to be either abbreviated or truncated.

It must be understood that these definitions do not describe *variables* of a program, but merely certain kinds of data that will be used. Such definitions are somewhat easily translated into a language like Pascal. It is not quite as simple to translate them into BASIC. However, such definitions can make a program much more understandable and can help to eliminate errors that are normally introduced because of confusion of various data items by the programmer. In addition, such definitions are not assignment statements, even though they may occasionally specify values. Their main purpose is to define the format of the data, not values.

Finally, note that there is no relationship between various data items, except as noted by the hierarchy, even when the definitions use the same data item names. The use of names in more than one place, such as in the case of *letter*, only implies that the data items being defined share a common format in part. This makes sense, considering, for instance, that we would want all dates in the system to have the same format. This consistency is necessary for creating an understandable and predictable system.

Example 2: A Carpet Store System

In the previous section, we developed a general description for a system that would help a carpet store owner develop and keep track of customer estimates and orders. In it, we defined at least two files, one for estimates and another for orders. Two other files might also be added in order to make certain calculations easier (this might not be obvious until later when the details of the logic are explored).

The first is a file that contains a code for each carpet the store sells, and an associated price of that carpet per square yard. The second is a customer file that contains information about every customer of the store. This file can be used to prepare mailings to customers, an added feature of this system that an analyst might suggest to the store owner. In addition, this file would contain the year-to-date purchases for each customer. This would be needed in order to calculate the discount each customer receives with any new purchase, and for defining a "regular" customer.

Figure 1.4-1 gives the data definitions for the estimate file. This is obviously a very lengthy set of definitions. It was created by simply looking at the estimate form and detailing all of the items that must be filled out by the salesperson in order to create an estimate. Some of the items, such as *customer-info,* have been provided with their own definitions in order to make the definitions more understandable. In addition, it makes it possible to use these definitions as part of other data items.

The exact format of certain data items is entirely arbitrary and could be defined differently. For instance, the *date* item could have been defined with dashes instead of slashes, or the names of months could have been used. The *state* item would perhaps have been more precisely specified by listing all fifty of the state abbreviations. However, such tedium can be avoided sometimes with simple English statements. It will still be necessary for these abbreviations to be fully specified at some point. A table copied from a zip code directory could be attached to the data definitions in order to make this easier.

Note that some definitions have a specification such as 3{thing}3. This means that there are to be exactly three *things* as part of this data item.

Finally, it should be noted that the definitions given introduce a certain type of error into the system. Note that the length and width of a room can each be as large as 100 feet, 11 inches. This means that the maximum area of a room is 10,404 square feet (102 * 102, since the dimensions must be rounded up to the next yard), or 1156 square yards. The maximum total cost of one room can be $99999.99. This implies that the maximum price of carpet per square yard could be $86.50 (99999.99/1156). This seems to give enough leeway for the room total, since few carpets, if any, will ever approach this price. However, we can have up to seven rooms. The cumulative total for all seven rooms must also be less than $99999.99. This means that the average room size and average price of carpet per square yard must be low enough so that the total dollar amount for all rooms does not exceed $99999.99. This would occur, for instance, if there were seven rooms to be carpeted, all approximately 100 feet by 100 feet in dimension, and the average cost of the carpet for each of these rooms was greater than $12.37 (86.50/7).

```
Estimate-file = {estimate}
estimate = customer-info
         + date
         + 1{room-info}7
         + salesperson
         + net
         + discount
         + subtotal
         + tax
         + total
customer-info = billing-name
              + billing-address
              + (contact-name)
              + (phone)
billing-name = [person-name | company-name]
person-name = first-name
            + "b̸"
            + middle-name
            + "b̸"
            + last-name
first-name = 1{letter}10
letter = ["A" . . "Z" | "a" . . "z"]
middle-name = [{letter}10 | letter + "."]
last-name = 1{[letter | "b̸"]}20
company-name = 1{letter | digit | "b̸" | "-"}30
digit = "0" . . "9"
billing-address = address
address = street
        + city
        + state
        + zip
street = 1{digit | letter | "b̸"}30
city = 1{letter | digit | "b̸"}20
state = 2 letter state abbreviation
zip = [9{digit}9 | 5{digit}5]
contact-name = person-name
phone = 3{digit}3
      + "-"
      + 3{digit}3
      + "-"
      + 4{digit}4
date = "01" . . "12"
     + "/"
     + "01" . . "31"
     + "/"
     + "84" . . "99"
room-info = width
          + length
          + square-yards
          + carpet-code
          + square-yards-price
          + padding-price
          + installation-price
          + room-total
```

FIGURE 1.4-1: A data dictionary for the Estimates File of the carpet store application. In this and subsequent figures, "b̸" indicates a blank or space. The list of data items is given in a top-down fashion.

```
width = dimension
dimension = feet
          + inches
feet = 0 . . 100
inches = 0 . . 11
length = dimension
square-yards = 0 . . 1156
carpet-code = "1" . . "9"
            + 4{digit}4
square-yards-price = dollars
dollars = floating point number between 0 and 99999.99,
          inclusive, with exactly 2 decimal places
padding-price = dollars
installation-price = dollars
room-total = dollars
salesperson = person-name
net = dollars
discount = dollars
subtotal = dollars
tax = dollars
total = dollars
```

FIGURE 1.4-1: ***continued***

While it is perhaps highly unlikely that this will ever occur in a real-life situation, it must be realized that such cumulative errors can frequently cause havoc in a system. They should be avoided at all cost by placing reasonable ranges on all values, or by ensuring that the precision of a data item is enough to handle the largest possible values. In this case, the *total* data item should be able to handle seven times the largest possible amount for a room total, plus the largest possible tax amount. In this case, this would require that the precision of the *total* data item be at least an order of magnitude larger than the precision of the *room-total.*

Once the data definitions for estimates have been completed, we can move on to defining the other files. Figure 1.4-2 gives the definitions for the orders file. Note that these definitions draw heavily upon the definitions previously developed for the estimates file. This is natural, since the estimate form and the order form are nearly identical. The *order* definition adds the items that are filled in for an order but not for an estimate.

Figure 1.4-3 gives the definitions for the carpet file that contains a table of carpet codes and their per yardage prices. Again, since *carpet-code* and *square-yards-price* were previously defined in Figure 1.4-1, this definition is complete.

```
Order-file = {order}
order = estimate
        + order-number
        + installation-address
        + installation-date
order-number = 5{digit}5
installation-address = address
installation-date = date
```

FIGURE 1.4-2: A data dictionary for the Order File of the carpet store application. This portion of the application's dictionary refers to data items defined previously in the dictionary in Figure 1.4-1.

Finally, Figure 1.4-4 gives the definition for the customer file. Note again that a cumulative error could crop up here, since the *year-to-date-purchases* has been defined as the same precision as the *total* data item. The question that cannot be easily answered is What maximum number of purchases is a customer likely to make in a year's time? Some assumption would have to be made about this. This assumption would then be incorporated into the required precision for the *year-to-date-purchases* data item.

This concludes the definitions required for describing the files. What about other data that will be needed in the program, much of which will be individual items not connected with files? Such data items must also eventually be added to the data dictionary, since the dictionary will become a valuable part of the documentation when the program has been completed. However, these additional items may not be known until later in the life cycle of the program, such as in the design or implementation phases. Whenever during the program's development you find the need for a data item that has not yet been defined, add it to the dictionary. Such definitions can take advantage of any previous definitions, just as was done in the examples above.

FIGURE 1.4-3: A data dictionary describing the Carpet File for the carpet store application. Note that the definition specifies that this file must contain at least one record.

```
Carpet-file = 1{carpet}
carpet = carpet-code
        + square-yards-price
```

```
Customer-file = {customer}
customer = customer-info
        + year-to-date-purchases
year-to-date-purchases = dollars
```

FIGURE 1.4-4: A data dictionary for the Customer File for the carpet store application.

Data Flow Diagrams

In a large project that uses a great deal of data and has many files, it is often difficult to visualize how all the various pieces fit together. This is especially true in systems with many programs, even though these programs are obviously related to one another. The fact that these programs may share data through files makes the task of keeping track of what goes where even more difficult. The consequences of not being able to keep track of this information can be quite devastating. The most common problem occurs when a file's format must be changed. Without some type of guide to tell which programs use that particular file, a program that should be changed might not be, resulting in an error that is very difficult to trace later on.

Data flow diagrams are used to overcome this problem. Their main purpose is to identify what programs in a system use which data. There are many forms of such diagrams, some more complex and detailed than others. The one presented here is, I feel, the most straightforward, and provides sufficient detail to do the job well.

The diagrams are not going to be concerned with individual data items, as was the case with the data definitions. They are instead concerned with large collections of data, sometimes called **data sets.** These data sets come in three main forms: data files, manual inputs, and printed outputs. There are categories that can be defined within each of these formats, but there is usually little to be gained by going into any greater detail. For instance, data files might be stored either on tape or disk, on floppy disk or hard disk. Such distinctions can, for the most part, be safely ignored.

Finally, the data flow diagrams describe which data each program in a system uses. The programs of the system can be obtained from the HIPO charts that define the hierarchy of program functions in a system. However, since even HIPO charts may change during the life of a project, the data flow diagrams must be kept up to date. Like most of the other techniques discussed in this book, these diagrams should be viewed as dynamic, changing during the life cycle as more details are known about the system.

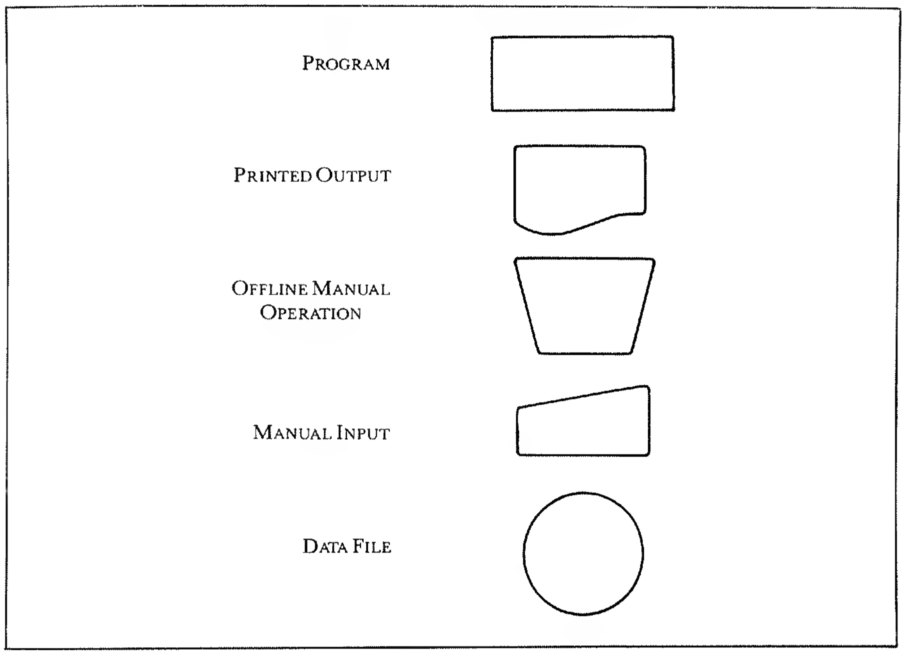
The Diagram Symbols

As mentioned above, there have been many types of data flow diagrams developed, for several different purposes. The diagram defined here is used to present a description of how collections of data are used throughout the application system. The symbols it uses are based on those used in flowcharts, which are discussed in Chapter 2.

Figure 1.4-5 shows the symbols used in a typical data flow diagram. The *program* symbol simply indicates the functions that will be performed by a program. The name of the program is placed in the box. This name is also associated with a general description of the program's purpose.

The *printed output* symbol indicates any type of output that is printed. This includes the use of special forms, such as checks, as well as simple reports. This symbol does *not* indicate data that is displayed to the user of the program using a device such as a video

FIGURE 1.4-5: The symbols used for drawing data flow diagrams.



display. Unless the screen is the main output device, such outputs can be included in the diagram using another symbol, but such additions tend to clutter the diagram and add little to the meaning of it. We will, therefore, ignore any outputs of this kind in the diagrams.

An *off-line manual operation* is one that is performed by hand, not in direct connection with the computer. Filling out a paper form, for instance, would be such an operation, and is, perhaps, the most common task identified with this symbol.

Next, a *manual input* is any input that is performed directly by a human being. This would include typing in data, pushing buttons, or using a joystick. For most applications, this will simply mean data that is entered by the user via a keyboard.

Finally, the *data file* symbol is used to indicate all types of data files. This symbol is independent of the medium being used to store the information, such as tapes, floppy disks, or hard disk devices. Purists have a different symbol for each medium, including punched cards and paper tapes. This certainly adds information to the diagram. However, there is a lot to be said for simplicity in diagrams that are often quite naturally complex. Therefore, we will use this single symbol for all data files.

Figure 1.4-6 shows an example of a data flow diagram for a system that is used by a manufacturing company. The system is made up of seven programs. Associated with each program is one or more collections of data that are used by that program in some way.

The flow lines indicate whether a data set is an input, an output, or both, by the direction of the arrow. An arrow into the program indicates an input. If the arrow is pointed into the data set, it is an output. This output may be simply a modification to the data, and does not necessarily indicate that the entire data contained in the collection has been generated by the program.

For the *Order Entry* program, a salesperson creates an order form by hand, which is then entered into the program by an entry clerk. The program also uses the *Product File*, which is used in addition by the *Marketing* program. This type of link is common throughout the diagram, where the output of one program becomes the input of another. This indicates an order to the execution of the various programs of the system as far as the data is concerned. For instance, the *Order File* produced by the *Order Entry* program must be created before the *Inventory Control* or *Marketing* programs can be executed. Therefore, the *Order Entry* program must be executed first.

This does not mean that the programs cannot both be executing at the "same time" in a time-sharing system, i.e., that the order entry

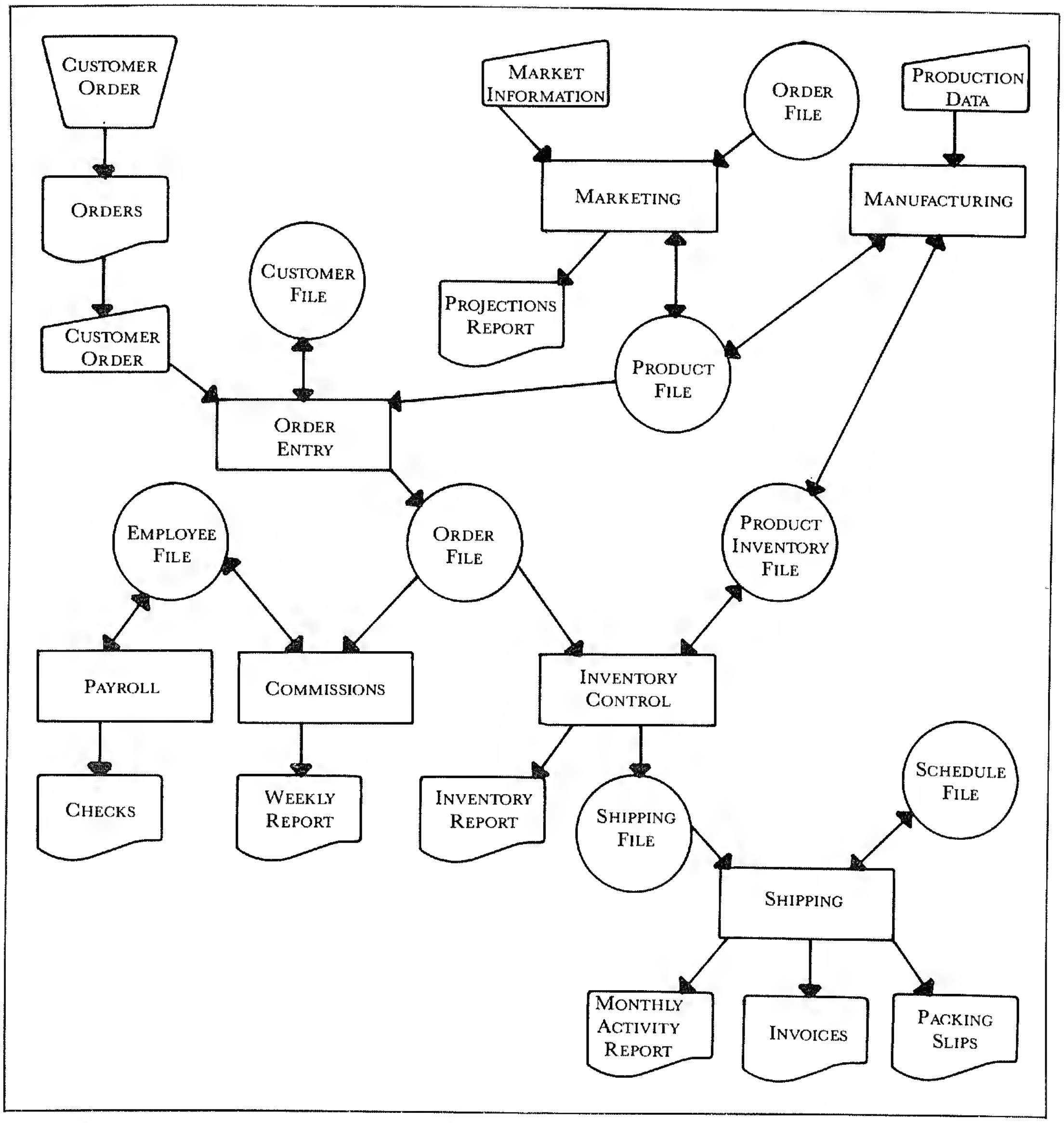


FIGURE 1.4-6: **An example of a data flow diagram for a system that might be used by a manufacturing company.**

clerk cannot be entering orders at the same time the marketing staff is running a new projection for next month's sales. What it does mean is that the programs are probably operating on different versions of the same data set. The marketing people are probably using yesterday's data, while the order entry clerk is creating today's order file.

Consider how it would work if the three programs *Order Entry, Inventory Control,* and *Shipping* are all scheduled to run exactly once each day, and they are all three begun at the "same time." The *Order Entry* program takes all day to run, so that the *Order File* is not ready to use until the end of the day. The *Inventory Control* program does not wait until the *Order Entry* program is finished generating a new *Order File* in order to begin its work. Instead it uses yesterday's file, since the data in that file has not yet been used by the *Marketing* program. The *Shipping* program, likewise, cannot wait for the *Inventory Control* program to finish, so it uses the *Shipping File* that was also generated yesterday. But yesterday's *Shipping File* was generated using the day before's *Order File.* As a result, the information generated by the *Shipping* program is two days old as far as the current batch of orders is concerned.

This is a fairly common mechanism for scheduling programs using multiple versions of data sets. It can be quite confusing trying to keep track of which version each program uses. Unfortunately, the data flow diagrams do not easily provide a method for keeping track of such details. This is even more obvious when looking at the data sets that are not only shared by multiple programs, but are also both input and output by each of the sharing programs. The *Product Inventory File* is an example of this. Which program created the version that the two programs are using? What is the order of usage? Should *Manufacturing* use the data generated by *Inventory Control,* or the other way around?

This problem has been greatly alleviated by the creation of **data bases,** data sets that can be shared among several users simultaneously. This eliminates multiple versions of each data set, so that every program is accessing exactly the same set as any other program. This technique could also eliminate the time lag between when data is entered and when it is useable, as in the case of orders in the example. This concept is being put into operation in most large-scale installations today. This mechanism is often much more powerful than is required, however, and is usually reserved for quite complex systems such as that shown in Figure 1.4-6.

The Carpet Store Revisited

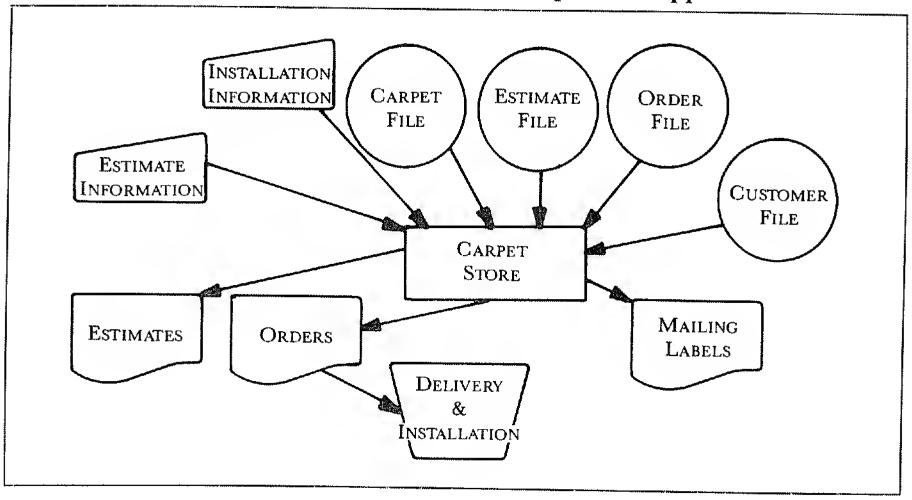
Let's return now to the carpet store system we've been developing. We would like to create a data flow diagram for this system. We know several of the files that will be needed, and we can add the various manual inputs and printed outputs that will be part of the system.

Figure I.4-7 gives a first attempt at creating the data flow diagram for this system. It shows the various files used as both inputs and outputs. It also shows estimate and installation information being entered manually into the system. Finally, the program generates estimates, orders that are turned over to the delivery and installation crews, and mailing labels.

But how useful is this diagram? It does include some information that hasn't been explicitly presented before, showing the manual inputs and the various forms that are going to be generated. However, this is not a thorough enough view of the system to do us much good.

In the previous section, we saw that the carpet store problem can be thought of as a system made up of several programs. The main part of the system can be viewed as two separate programs, even if these are not implemented as physically separate pieces of code later on. As shown in Figure I.4-8, the first part is the *Salesperson Estimate* program, which takes the *Carpet File* and *Customer File* as inputs, along with the *Estimate Information* that is collected from the user and entered manually by the salesperson. This program generates the *Estimate File,* containing a copy of the customer's estimate that also is printed out.

FIGURE 1.4-7: A first attempt at a data flow diagram for the carpet store application.



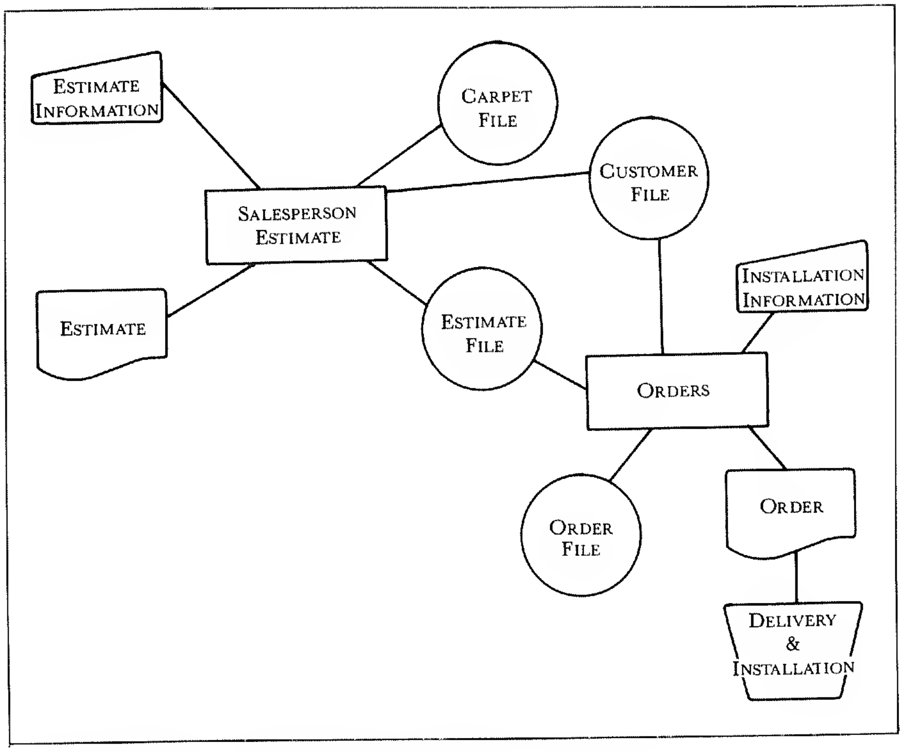


FIGURE 1.4-8: A data flow diagram for part of the carpet store application.

The *Orders* program takes this new *Estimate File* as input, along with the *Customer File* and the manually entered *Installation Information.* It generates the *Order File* and a printed order form, which is used by the delivery and installation crews.

The complete system was discussed in more detail earlier in this chapter (1.3 Modular Design). It is left as an excercise for the reader to complete the data flow diagram based on this design of the system.

2 Structured Design Concepts

2·1 INTRODUCTION

Developing a program can, in many ways, be likened to writing a book. The systems analysis phase can be viewed as the author's process of gathering notes and references. The second stage, designing the program, is like outlining the book's chapters. Summarizing the content of the book's chapters is akin to developing outlines of modules based on the definitions and notes prepared during the analysis phase.

During this second stage, we are not concerned with the details of the program's implementation. While a programmer might feel compelled or inspired to begin coding a module immediately after its definition in the analysis phase, such temptation must be fought. Invariably, giving in to such temptation ends in a program that is poorly structured and, therefore, difficult to maintain.

The techniques discussed in this chapter may appear to introduce redundancy into the development process. Indeed, since the design of a program will be done in its own special-purpose language, called **pseudo-code,** it will appear that the program is actually coded twice. However, the **algorithm** that is designed using this pseudo-code is closer to English than to code. It summarizes the actions that are to be taken in the program but does not introduce the details of the implementation of these actions in a particular programming language.

This accomplishes several key things. First, it makes each module's design independent of any real programming language. In this way, the program could easily be written in any programming language. The design of the program should not depend on whether it is finally to be implemented in BASIC, Pascal, FORTRAN, COBOL, assembly language, or some other. The further the design is kept from the implementation language, the more adaptable it is. In addition, such insulation tends to strengthen the correctness of the implementation.

Second, it is far easier to change an algorithm written in pseudo-code than to change a module coded in a programming language. Again, the analogy to writing a book is appropriate. The design of a program is an iterative process and requires many drafts, just as a book would. The use of algorithms and pseudo-code is like the first draft of a manuscript; heavy editing will certainly be needed before the final draft is made. In the case of programming, the implementation itself can be thought of as the final draft.

Third, using these techniques allows a certain amount of experimentation to take place. There is generally no one best solution to a programming problem, just as there is no one best design for an automobile. Alternate approaches can be tried with minimal cost in time and material using algorithms and pseudo-code.

Finally, this type of designing allows the various approaches to be evaluated through a testing procedure (algorithm and program testing are discussed in more detail in Chapter 5, Program Testing and Debugging). Only the more promising designs are pursued further. The alternate designs can be discarded without concern for the cost, since the cost is minimal in comparison to the cost of code.

In many ways, developing algorithms is like building models for architects or engineers. If the design doesn't work, only the model suffers. It is much more costly to repair a fault in a fully built system than in a model. Models are worked and reworked until the engineers are satisfied that the design makes sense. From there, blueprints are developed from which the final implementation is made.

We can consider the models to be the algorithms developed for each module in a program. Flowcharts, a graphical form of algorithms, could be considered the blueprints for the implementation phase of the project. No design engineer would attempt to build a production system without models and blueprints. No professional programmer would attempt to implement a sophisticated piece of software without at least developing detailed algorithms.

2·2 THE USE OF PSEUDO-CODE

Part of the reason that some time should be spent developing the details of the logic of a program before actual coding begins is that programming languages possess certain characteristics that do not always lead to well-implemented programs. The most notorious of these is the GOTO statement, which many computer scientists believe to be the root of all evil when it comes to bugs in programs. Using GOTOs indiscriminately in code leads to the "spaghetti bowl" syndrome; i.e., it is impossible to follow program logic when the code is full of GOTOs leading from one section of code to another, perhaps several pages away, just as it is impossible to follow a single strand of spaghetti from beginning to end in a spaghetti-filled bowl.

The problem is that there is often no choice but to use the GOTO statement to alter the flow of a program's logic, simply because of the nature of the particular programming language. For instance, most implementations of BASIC and FORTRAN could not survive without the GOTO. These languages usually do not support the more complex logic constructs that make it possible to excise the GOTO from use.

There are similar problems with other characteristics of certain programming languages. However, probably more than ninety percent of all programming is done in languages with these problems—languages that do not adhere to the structured precepts we would find most beneficial. So we have a great dilemma in which we must decide either to forsake our favorite programming language for one which does incorporate the structure desired, or to violate structured programming philosophies.

To add even more confusion, there are times when only a particular language will be appropriate for an application. For instance, sometimes it is necessary to write a program in assembly language, which is obviously the most unstructured language in existence.

Do we simply abandon the wonderful structured precepts that have proven to be effective in eradicating bugs from our programs? Luckily, pseudo-code provides us with a better option. A high-level pseudo-code can be defined which contains all the control structures required for developing a well-formed program. Because pseudo-code is not a real programming language, there is no compiler for it. But we can treat it as a real language while we are still in the design phase of our program or system.

An algorithm developed using pseudo-code is simply a set of instructions that will be followed, step-by-step, in order to perform some task. It differs from a program in that it is expressed in a natural language such as English, not a programming language. However, English by itself is not adequate for the job because of its penchant for ambiguity. Therefore, several standard structures have been defined that control the sequence of execution of the steps in the algorithm.

Pseudo-code plays several important roles during program design:

1. It allows us to present the design detail of a program or module at a high level using English-like statements, thus still avoiding low-level logic issues, such as input and output considerations.
2. It provides a process which forms a foundation on which to build structured programs, since adherence to the formal constructs enforces structure.
3. It acts as a bridge between the high-level design developed using the techniques of Chapter 1 and the actual implementation details that will be of concern in Chapter 3.
4. It provides one more link in the documentation chain.

There is only one real problem with pseudo-code: there is no standard form for it. While there are usually minimal constructs for pseudo-code, there have been a number of extensions developed. In addition, allowing the actual instruction to be written in English means that what one programmer uses to indicate a certain instruction will be different from what another programmer uses.

Let's look at the various control structures usually defined for a pseudo-code, and then readdress the issue of how to keep the style both simple and understandable.

Sequence

The simplest form of statement ordering is the **sequence,** which is a series of statements that will be performed in a linear order. For example:

```
get into the car
drive to the grocery store
select groceries
buy groceries
get into the car
drive home
unpack the groceries
```

We begin executing these instructions with the first one, namely *get into the car,* and continue with the next in the list, advancing one at a time until all statements in the list have been performed. Some purists would insist that each statement be ended with a semicolon, so that each individual statement is more obvious. This is particularly helpful when more than one statement appears on the same physical line, such as

```
turn off the alarm; take a shower; get dressed
```

Here the semicolon acts to separate one statement from another. This "rule" can be followed somewhat informally, however, and used only when necessary, since we don't have to worry about a compiler becoming upset over changes in syntax.

It is sometimes necessary to be even more explicit in identifying a group of statements that should be taken together as a single block. In this case, we can use special keywords to indicate which statements are to be grouped together. For instance:

```
BEGIN
   turn off the alarm;
   take a shower;
   get dressed
END
```

The purpose of this blocking will become evident when we discuss the other control structures below. Notice that the keywords BEGIN and END are written in uppercase letters, while the statements themselves are in lowercase. It is usually helpful to be able to easily identify keywords (words that are a part of the pseudo-code language) within an algorithm, so they are usually highlighted in some way. Many authors seem to prefer boldfacing the keywords, others underline them. Capitalizing is a good method even when pseudo-code is handwritten, not word-processed. Finally, indenting any instructions specified within the BEGIN-END block helps to highlight the block and adds to the readability of the code.

Selection

We are often faced with situations that call for making choices, or selections, in a program. In these cases, we need to be able to decide on executing one group of statements or another based on some condition(s). The choices that will have to be made range from the trivial to the very complex. One statement in particular, the IF-THEN-ELSE construct, is used in the vast majority of these situations. A more intricate mechanism, the CASE statement, is used when there are more than just a couple of alternatives to select from.

IF-THEN-ELSE

We can ask questions and define alternate actions based upon the answers using the IF-THEN-ELSE construct. For instance:

```
IF trying to lose weight
   THEN take the steps
   ELSE take the elevator.
```

Here we have two well-defined alternatives. The action we take depends on the answer to the question. If the answer is true (or yes), then we perform the actions defined by the statements following the THEN keyword (often called the **true clause**). If the answer is false (or no), then we perform the actions defined by the statements following the ELSE keyword (or the **false clause**). The questions asked must, therefore, be binary in nature, i.e., having at most two possible answers, either true (yes) or false (no). The general form of the IF-

THEN-ELSE can be shown as

```
IF ⟨binary condition⟩
   THEN ⟨true statements⟩
   ELSE ⟨false statements⟩.
```

If we want to indicate a group of statements being executed in either the true or false clause, we can use BEGIN and END to block them.

```
IF this is the weekend THEN
BEGIN
    take a nap;
    mow the lawn;
    party
END
ELSE work
```

Indenting and placing phrases on separate lines are used for emphasis and to aid readability by visually indicating the scope of a block. The exact format for this *pretty printing* (as it's called in the business) is entirely up to the programmer.

A particular decision may require an action only in the case when the condition is true. In this case, the ELSE clause is simply eliminated.

```
IF no food in the house for dinner
   THEN order a pizza delivered
```

The ELSE clause is thus optional. If the condition is false, then the next statement following the IF statement is executed, just as in any sequence.

```
IF the gas gauge shows a low tank
   THEN get gas;
drive to the convention
```

Note the semicolon at the end of the THEN clause. This is because the IF-THEN or IF-THEN-ELSE construct counts as a statement itself. This should make it clearer that the *drive to the convention* statement is not a part of the THEN clause. The indentation should also make this clear. If there seems to be any ambiguity in this, use the BEGIN-END block to partition the statements.

```
IF low on gas THEN
BEGIN
    find a gas station;
    get gas
END;
drive to the convention
```

There isn't a semicolon after *get gas* this time. We don't need one since it is followed by END. However, the END keyword needs a semicolon here because it separates the IF-THEN statement from the *drive to the convention* statement. Again, don't be overly concerned about getting this exactly right.

Although the ELSE clause is optional, the THEN clause isn't. For instance, the following is *not* valid:

```
IF older than 16
   ELSE go to school
```

If there is ever a temptation to construct such a statement, simply negate the condition and replace the ELSE clause with THEN:

```
IF not older than 16
   THEN go to school
```

Another option for creating IF statements is that the conditions themselves can be complex, combining more than one condition for deciding whether some statements will be executed. Additional conditions can be combined using standard conjunctions AND and OR, which work just as they do in English.

```
IF mom is home OR dad is home
   THEN keep stereo turned down
   ELSE blast the neighborhood;
IF mom is home AND dad is home
   THEN do homework
```

Here we have two IF statements combined in a sequence. In the case of using the OR conjunction, if either of the conditions is true, then the THEN clause is performed, otherwise the ELSE clause is performed. In the case of AND, both conditions must be true in order for the THEN clause to be executed.

Finally, IF statements can themselves appear as statements within either the THEN clause or the ELSE clause of another IF statement. This is called **embedding, nesting,** or **compounding** IF statements.

```
IF today is Monday THEN
BEGIN
   IF it is in the AM
      THEN drink lots of coffee;
   take long breaks
END
```

There is a danger in embedding too many IF statements, however. In particular, the logic gets somewhat difficult to follow for even the best programmer if nesting gets more than about three IF statements deep. The only hope of avoiding this situation is to redesign the logic so that the nesting is eliminated.

CASE

If the type of nesting is such that the conditions of the nested IFs form a group of single alternatives, a different control structure for selection can be used. It is called the CASE statement, and offers a way of choosing between several mutually exclusive alternatives. For instance, if we were to have the following situation:

```
IF today is Monday
   THEN do one thing
   ELSE
      IF today is Tuesday
         THEN do another
         ELSE
            IF today is Wednesday
               THEN do a third
               ELSE . . .
```

This would cause nesting six levels deep just to account for different processing for each day of the week. What if we need different processing for each month of the year? The CASE statement provides a neater way of doing the same thing.

```
CASE day of week OF
Monday:      do one thing;
Tuesday:     do another thing;
```

```
Wednesday:  do a third thing;
Thursday:   do a fourth thing;
Friday:     do a fifth thing;
Saturday:   do a sixth thing;
Sunday:     do a seventh thing
ENDCASE
```

Here, the **case selector** is the day of week. We then enumerate all the possible options for this selector, being certain that each option is in fact unique. When a particular case is selected, any statements specified are executed, and then the next statement following the ENDCASE keyword is executed.

A case option can have several different selection values, as long as *all* options are unique in the CASE statement. For instance,

```
CASE day of week OF
Monday, Tuesday:     BEGIN
                         do several things
                     END;
Wednesday, Thursday, Friday:     do something;
Saturday, Sunday:     do another thing
ENDCASE
```

However, the case options *must* be mutually exclusive, i.e., there cannot be any overlap of selections between one option and another. The following is therefore illegal:

```
CASE day of week OF
Monday, Tuesday, Wednesday:     do something;
Wednesday, Thursday, Friday:     do another something;
Saturday, Sunday:     do some other thing
ENDCASE
```

The first and second options have "Wednesday" in common. We can only have *one* of the options selected whenever a particular case is present.

Note in the previous examples that the cases enumerated give all possible options for this particular case selector. There cannot be any *day of week* other than those presented as cases, unless there is an error somewhere. This is not always the situation. In addition, it might be helpful to have some standard case available that gives us an alternative to *all* other enumerated cases. An OTHERWISE case would give us this alternative.

```
CASE day of week OF
Saturday:          mow the lawn;
Sunday:            BEGIN
                     go to church;
                     relax
                   END;
OTHERWISE:         go to work
ENDCASE
```

The fully general form of the case statement is as follows:

```
CASE ⟨case selector⟩ OF
⟨case #1⟩:         ⟨statements for case #1⟩;
⟨case #2⟩:         ⟨statements for case #2⟩;
                              . . .
⟨case #n⟩:         ⟨statements for case #n⟩;
OTHERWISE:         ⟨statements for all other cases⟩
ENDCASE
```

Iteration

It is fairly obvious that there is some need for a construct that will allow us to specify that a group of statements be executed a number of times, i.e., repetitively. In common programming parlance this is called a **loop.** What is not obvious to even some experienced programmers is that there are several types of loops, and that selecting the proper type for a particular purpose is very important. Using the wrong loop is just the kind of logical error that gives programmers nightmares. The problem usually doesn't show up until months after the program was tested and accepted for use. Then, even after the bug does manifest itself, trying to locate the actual error is often extremely difficult, since the code "looks" right.

WHILE

The mainstay of the loops is the WHILE statement. Its general form is

```
WHILE ⟨condition is true⟩ DO ⟨statement⟩
```

Here, a condition is tested first. If the condition is true, then the statement following the DO keyword is performed. We then return to look at the condition again, then perform the statement again, etc., until we reach a point when the condition tests out false. When this happens, the next statement in sequence following the WHILE is executed. For example:

```
WHILE the temperature is greater than 90 DO
        stay in air conditioning
```

There are several things to keep in mind about WHILE loops.

1. More than one statement can form the **body** of the loop, i.e., can follow the DO keyword, simply by grouping the statements using a BEGIN-END block.
2. The condition of the WHILE must be binary in the same way as the condition used in IF statements. In addition, the condition can be complex, using AND or OR.
3. The testing of the condition of a WHILE loop happens *first,* i.e., before any of the statements of the loop body are executed. As a result, it is possible that the body of the loop might never be executed because the condition tests out false to begin with. Notice in the previous example, if the temperature is already less than or equal to 90, then the loop body will not be performed. This characteristic is very important to keep in mind.
4. The condition is checked to see if the loop is to be *continued.* As long as the condition is true, the body of the loop will be performed.
5. There is the possibility of creating the infamous **infinite loop,** i.e., a loop in which the condition will never be false. Again in the above example, if the temperature is never below or equal to 90, we will never get out of the air conditioning. While any possibility of having an infinite loop may seem remote, the prudent programmer takes extra precaution when setting up a loop to keep one from occurring. However, sometimes an infinite loop might actually come in handy. Consider the customer estimate function in our carpet store example. It is likely that this would be an endless loop, returning to the beginning of the program after every estimate is given. The loop might look like:

```
WHILE more customers DO perform customer estimate
```

As long as customers keep coming in, the customer estimate routine will continue to loop. This could be "forever," if the store is open twenty-four hours a day. A way of forcing an infinite loop is to use a condition that will always be true, such as:

```
WHILE 0 = 0 DO something
```

In this case, *something* is done forever because 0 will always be equal to itself.

There are certain situations where having the condition for continuance of the loop checked first can be a problem. Consider a loop that is to check a heat sensor as part of a fire alarm system. What we would perhaps like to say is:

```
WHILE temperature is less than 150 DO
    check the temperature sensor;
sound the alarm
```

What we want is for the alarm to be sounded as soon as the temperature goes above 150 degrees. As long as the temperature stays less than or equal to 150, we stay in the loop checking the sensor. So what's the problem? When is the temperature sensor first checked? The first time we get to this loop in a program (maybe this is the entire program, by the way), we try to look at the temperature without having checked the sensor.

It's like trying to guess the time without looking at your watch. You might be right, but what are the consequences if you are wrong? One solution to this dilemma would be to initialize the temperature to be less than 150, thereby guaranteeing that the loop body (*check the temperature sensor*) is performed. The only problem is that there might be times when this won't work properly, because of certain constraints of the programming language you're using. In addition, this is really a "patch," i.e., something that works but is not the most natural way of doing the task.

REPEAT-UNTIL

The real problem here is that the condition is checked *before* the loop body can be executed. What we need in this instance is a loop that performs the body *first*, then checks the condition to see if the body should be repeated. This is called the REPEAT-UNTIL loop, and would look as follows for the same alarm example as above:

```
REPEAT
    check the temperature sensor
UNTIL temperature is greater than or equal to 150;
sound the alarm
```

Here we use a loop where the condition is specified at the bottom of the loop. There are several differences between a WHILE loop and a REPEAT-UNTIL loop:

1. You can place as many statements as you want between the REPEAT and UNTIL keywords, without using a BEGIN-END block. That's because these keywords act as a natural block in identifying the scope of the statements involved in the body of the loop.
2. The condition specified in the UNTIL section is exactly the opposite of the one specified for the WHILE statement. This is because the condition is checked to determine if the loop should be *ended* in the case of the REPEAT, whereas with the WHILE statement, it is checked to determine whether the loop should be started or continued.
3. Since the condition is not checked until the bottom of the loop (after the body of the loop has been specified), this loop body will always be executed *at least once*. Recall that the WHILE loop body might never be performed.

Other than the above differences, the REPEAT loop condition is treated the same way as the WHILE loop, i.e., the condition is binary and may be complex. In addition, the caveat about infinite loops applies to the REPEAT loop. In the above example, what happens if the sensor is somehow stuck at 100 degrees?

There is often a need for a third type of loop. Consider the following:

```
I := 1;
WHILE I < 10 DO
BEGIN
    do something;
    I := I + 1
END
```

The *do something* in this loop will be performed exactly ten times. The variable I is used to control how many times the loop will be executed. There are three important parts to this type of loop: the initialization of the loop control variable (I in the above example); the testing of a continuation condition; and an increment of the loop control variable. This combination of statements is so common that most pseudo-codes include a loop specifically intended to perform this kind of task more easily than the WHILE loop above.

FOR

The FOR loop is a special form of the WHILE loop, in which an index variable is "automatically" initialized, tested, and incremented. A FOR loop to accomplish the same as the above would look like:

```
FOR I := 1 TO 10 DO do something
```

The general form for this loop is

FOR i := x TO y DO ⟨statement⟩

where i is the **index** or **loop control variable,** x is an expression for the **initial value,** y is an expression for the **test condition value,** and ⟨statement⟩ is a single statement or a compound (BEGIN-END block) statement.

A variation of this is a FOR loop that will count down instead of up. Its general form is:

For i := x DOWNTO y DO ⟨statement⟩

In this case, i starts out as a larger number and is decremented until it is less than the value to which the expression y evaluates.

There are several things to note about this loop:

1. Initialization is done when the loop is entered the first time, i.e., when the FOR statement is first executed. The initial value can be anything, even negative.
2. The value following the TO keyword is used to construct a condition for continuing the loop, i.e., as long as the index variable is less than or equal to this value, the loop is continued. In the case of DOWNTO, the test will continue the loop as long as the value of the index variable is greater than or equal to the test value.

3. "Standard" use of this loop assumes a constant increment of 1. This could be embellished by adding a STEP feature to the FOR statement that would specify the increment amount (e.g., FOR I := 1 TO 10 STEP 2 DO something).
4. The initial and test values can be complete expressions, including variables. For instance,

   ```
   FOR I := X to Y + 3 DO something
   ```

   is perfectly legitimate. When this is the case, however, the expressions are evaluated *only* when the FOR statement is first encountered, and are not reevaluated during the execution of the loop. For instance, consider:

   ```
   J := 10;
   FOR I := 1 TO J DO
   BEGIN
      do something;
      J := 12
   END
   ```

   This loop will still only be executed ten times, not twelve. Even more important is that the loop control variable (I) cannot be changed within the loop. For instance:

   ```
   FOR I := 1 TO 10 DO
   BEGIN
      do something;
      I := I + 3
   END
   ```

   is not legal. Therefore, never change variables used in the FOR statement while inside the loop body, since all it does is make it more difficult to read the code.
5. Since the FOR loop is based on the WHILE loop, the loop body might not ever be performed. This situation occurs whenever the initial value is already greater than the test condition value.

It cannot be noted too strongly that any apparent similarity between the FOR loop specified here and the FOR-NEXT loop of a dialect of BASIC is merely superficial. It should not be assumed that they are the same. One major point of difference may be **5.** above; some versions of BASIC base the FOR loop on the REPEAT loop, not the WHILE loop. In addition, several versions of BASIC allow the programmer to change the values of any variables used to control the FOR loop, such as the loop control variable. Some authors even encourage techniques that require such an action. This is discussed in more detail in Chapter 3, Structured Programming. For now, forget what you might know about FOR-NEXT loops in BASIC and get familiar with the FOR loop above.

Finally, loops can be embedded just as the IF or CASE statements can be. This means that any loop can appear within the body of any other loop. For instance:

```
WHILE A ⟨ 5 DO
    REPEAT
        FOR I := -3 TO 17 DO
            do something
    UNTIL we're done
```

Here we have a FOR loop embedded inside a REPEAT loop, which is itself embedded within a WHILE loop. This can get pretty tricky to decipher, so use this technique with extreme caution.

2·3 ALGORITHM DESIGN

Returning to the book-writing analogy, it is as difficult to explain how to develop algorithms as how to write a book. There are certain techniques that many authors use to prepare themselves for writing, and many methods they employ while writing. It is possible to explore these techniques and methods in some detail. Unfortunately, this is seldom enough to explain *how* to write.

It is generally accepted that the only way one learns to write is to practice writing. The more experience you gain, the more likely there will be improvement in your writing skills. This is not an absolute. The absolute is that one does not become a good writer by *not* writing. Writing is an art that must be practiced, much as one practices a musical instrument. The more you practice, the more familiar you become with style and technique. These eventually will become second nature.

Practicing means two things. First, you must write frequently. Few authors become good by writing only one short manuscript every year or so. They become good by turning out hundreds, if not thousands, of pages annually. Second, you must learn to revise what you have written. This means being able to read what you have written with a critical eye. You must learn the subtle art of unambiguous communication by being willing to modify what you have written until it feels "right."

The problem with this approach to learning to write is that it takes a long time to succeed. Even worse, there are few ways that you can be *taught* to write, just as there are few ways that you can be taught to play a musical instrument. Invariably, one must practice one's art alone.

What can be taught are techniques that may make your practice more productive and fruitful. Points of technique can be learned from a master of the art. These techniques may improve the quality of your own performance. The master becomes a mentor, instructing you in how to *learn* to become a master yourself. This, of course, is the ultimate goal.

Note that the purpose of the mentor has been defined as "teaching the techniques of how to learn" a particular subject. This is certainly the purpose of this book, and others like it—to teach you how to learn programming skills. Unfortunately, teaching is a dynamic process, requiring constant feedback in order to be effective, and the medium of the book is static, giving no feedback. This makes teaching about subjects such as algorithm development very difficult at best.

One solution to this dilemma is to attend programming workshops, in the same way that one would attend a writer's workshop. There, you can receive the immediate feedback that will ultimately improve your performance. In addition, once some experience is gained from such a workshop, you should find the techniques discussed in books easier to comprehend and implement.

Fortunately, in the case of programming, there is another way to gain the experience required and to receive the feedback needed to improve. The computer itself makes an excellent mentor!

By implementing an algorithm in code and executing it, the resulting output of the program will tell you how well you have created the algorithm. If the program does not perform its function properly, then in many cases the algorithms should be redesigned. This iterative process creates a feedback loop that will ultimately improve your ability to develop algorithms.

In this section, then, we will look at some of the standard techniques for algorithm development. Most of the techniques discussed are little more than definitions, however. Several examples are provided, showing the iterative process of algorithm development, but this, in itself, cannot teach you exactly how to write an algorithm.

It is expected that you will develop your skill first by mimicking the examples given. Next, you should begin to experiment with variations of the forms you are familiar with. Finally, it will be entirely up to you to explore new forms in algorithm design. Through frequent practice, your skill will grow. Your success will be exhibited in well-written programs.

A First Example

Let's first look at an example of an algorithm for an activity taken from everyday life. In reality, we function by algorithms in order to perform many of our daily functions. Consider, for example, a recipe. This is a step-by-step explanation of the actions that must be taken in order to properly cook something. While our implementation of such algorithms may occasionally leave a lot to be desired, the directions they provide are, generally, helpful in accomplishing our goal.

Following is a recipe for making barbecue ribs (try it!):

Ingredients:

1 rack of pork ribs	1/4 cup lemon juice
3 tbls. chopped onion	1 cup ketchup
2 tbls. butter or margarine	1/2 cup finely chopped celery
2 tbls. granulated sugar	
2 tbls. vinegar	1 tsp. dry mustard
1 tbl. Worcestershire sauce	salt and pepper

Salt and pepper the ribs. Place ribs on center of grill. Coals should be pushed to the side, with a drip pan in between two piles. Cook covered for 75 minutes. During the last 20 minutes, brush on the barbecue sauce. The sauce can be made as follows: Melt the butter or margarine in a skillet. Sauté the onions until they are tender. Add the remaining ingredients. Cook for about 20 minutes. If you like slightly tangier sauce, add either 1/4 tsp. chili powder or 1/2 cup chili sauce.

Nearly anyone could follow this recipe and end up with a successful meal. This particular sauce may not be to everyone's liking, but following the directions faithfully will reproduce the result intended by the cook.

This recipe is very similar to a general description of the result we are looking for. It is not quite an algorithm in the form we would like, however. Let's convert the recipe into a proper algorithm using the pseudo-code described in the last section.

Our first attempt might be to think in terms of the highest level functions that we need to perform. In this case, it might be:

```
Make-Barbecue-Ribs.
BEGIN
   Make-the-Sauce;
   Cook-the-Ribs;
   Put-Sauce-On-Ribs
END;
```

Note that making the sauce has been specified prior to cooking the ribs. Although it is possible for two things to happen simultaneously in real time, this is generally not possible for a computer. We should specify only one thing to happen at a time in an algorithm. We would want to make the sauce first, because otherwise the ribs would be getting cold while we prepared the sauce. While this would still accomplish our goal, it would *not* accomplish it in the most desirable way. This is one of the important techniques of algorithm design—getting things to happen in a logical order.

Although the algorithm above specifies the actions necessary to have barbecue ribs for dinner, the specifications are not detailed enough to be followed by a barbecue neophyte. If we think in terms of modules, however, we can see that the above algorithm does provide us with the skeleton of the steps to be followed. If we treat *Make-the-Sauce* as a reference to another algorithm, we can build a separate algorithm for that action alone. This algorithm might appear as:

```
Make-the-Sauce.
BEGIN
   Melt the butter or margarine in a large skillet;
   REPEAT
        Sauté onions
   UNTIL onions are tender;
   Add remaining ingredients;
   IF you like tangier sauce THEN
      IF you have chili powder THEN
         Add 1/4 tsp. chili powder
      ELSE
         Add 1/2 cup chili sauce;
   REPEAT
        Place skillet over medium-low heat
   UNTIL 20 minutes elapse
END;
```

This obviously looks somewhat different from the original recipe. In order to unambiguously describe the actions, it is necessary to introduce control structures. While the original recipe might be understandable to human beings, it is necessary to use very careful phrasing when preparing an algorithm for a computer to execute.

Note that a couple of the statements had to be rearranged in order to make sense in the algorithm. In addition, look at the IF statements for deciding whether you want tangier sauce. The implication here is that the second IF statement won't be executed if you do not want tangier sauce. If you do want tangier sauce but do not have chili powder, this algorithm assumes you have chili sauce on hand. Such assumptions are not always reasonable. Perhaps a better way to express this would be:

```
IF you like tangier sauce THEN
   IF you have chili powder THEN
      Add 1/4 tsp. chili powder
   ELSE
        IF you have chili sauce THEN
         Add 1/2 cup chili sauce;
```

But what do you do if you have neither ingredient but you want tangier sauce? According to this code, you would add nothing to the sauce. An assumption that you have all ingredients on hand is perfectly reasonable in this case, but such assumptions are not always reasonable. Understanding what assumptions can and cannot be made when developing an algorithm is a difficult task.

The second module might be represented as:

```
Cook-the-Ribs.
BEGIN
   Prepare-the-Grill;
   Salt and pepper the ribs;
   Place ribs in center of grill;
   Place cover on grill;
   REPEAT
        Cook the ribs
   UNTIL 55 minutes elapse
END;
```

This is fairly straightforward. Note that, while the original recipe called for cooking the ribs for seventy-five minutes, this routine only cooks them for fifty-five minutes. The next module, *Put-Sauce-On-Ribs,* will be responsible for making up the final twenty minutes. In addition, consider the statement *Prepare-the-Grill.* This didn't occur in the original recipe. However, the action was obviously implied by the description. This is a third principle about algorithm design: you must discover the hidden implications of the actions that need to be performed.

The new module might be defined as:

```
Prepare-the-Grill.
BEGIN
   Make two piles of charcoal;
   Light charcoal;
   REPEAT
        Let charcoal burn
   UNTIL charcoal covered by grey ash;
   Place drip pan between piles of charcoal;
   Place grill over coals
END;
```

Since this last algorithm does not introduce any new modules, let's return to the higher-level modules in order to complete the specification of the entire algorithm.

The *Make-the-Sauce* and *Cook-the-Ribs* algorithms have now been fully specified. The only activity left undone is the module *Put-Sauce-On-Ribs.* This could be specified as follows:

```
Put-Sauce-On-Ribs.
BEGIN
   Brush sauce on ribs;
   REPEAT
        Cook the ribs
   UNTIL 20 minutes elapse
END;
```

Again, there are implications and assumptions that have entered the algorithm. It is necessary to split an action here (*Cook the ribs,* split as two time periods, first 55 minutes and then 20 minutes) in order to allow the statements to progress in an orderly manner.

In summary, the following principles should be kept in mind when developing an algorithm:

1. Think in terms of modules that perform specific functions. Use a top-down technique in defining each module's contents.
2. Statements must be analyzed to discover their proper logical order.
3. Any assumptions that may have been made about actions in an algorithm must be discovered and explicitly presented.
4. Any hidden implications of actions in the algorithm must be discovered and addressed explicitly.

More Mechanical Details

In section 2.2, The Use of Pseudo-code, we explored the control structures on which the logic of an algorithm should be based. There are other details that must be understood in order to adequately present an algorithm. Following are discussions of these additional details.

Variables

Had the previous example been an algorithm that we intended to implement on a computer, it would have been necessary to flesh out a few more details. For instance, how were we going to determine how much time had elapsed? This would have required introducing some counting mechanism, which in turn would have created a need for some variable to keep track of the time.

VARIABLE DICTIONARY It will be necessary to keep track of the values generated by calculations, even in algorithms where the code is very English-like. As a result, we will need to construct a **variable dictionary** that explains the purpose of each variable used in the algorithm. One dictionary for each module should be constructed.

The dictionary should contain a list of the names of all the variables used in one module. The name of a variable can be very informally constructed of English words that describe the variable's purpose. Dashes or periods can be used to string multiple words together to form a single name. In addition to a list of the variable names, the dictionary should contain a brief English definition of each variable's purpose.

What variables you will need for a particular module may not be immediately obvious. You will undoubtedly add to the dictionaries as you develop each module. Some of the variables will have been previously defined using the data definition technique. This will certainly form a strong base for the dictionaries.

COMPLEX VARIABLES Most variables will be either simple numeric variables or string variables, but it will be necessary to allow for more complex variables such as arrays and records. It is usually not necessary to be specific about the form of these **data structures.**

For now, we will treat arrays as either lists or tables of values. In referring to an element of these structures, we could use English descriptions of the elements' locations within the structure. Alternatively, we could use standard mathematical notation for referring to elements. Finally, we could use a form that is similar to the syntax of an actual programming language, although this could be somewhat confusing for someone not familiar with that language. For example, any of the following forms would be acceptable:

the 10th element of the list
the J^{th} element of list TEMPERATURES
TEMPERATURES_J
TABLE(I,J) or TABLE[I,J]
the element in the I^{th} row, J^{th} column of the table

In the case of files, it is usually safest to assume that all input and output is performed with records. We can, therefore, simply refer to ". . . the record of file A. . ." in an operation. Individual fields can be defined for the record in the variable dictionary. These fields can be referred to simply by name. If the same field can be associated with more than one record, it may be necessary to refer to the specific record to which a field belongs. This is most often the case when dealing with input records and output records for the same file. These references can be of the form:

set X to TIME from the input record of file A
set TIME in the output record of file A to zero

Input and output operations are discussed later in this chapter in the External Data section.

Internal Data

Any values generated by the program itself, as opposed to obtained from outside the program (e.g., from a file or a user), will be called **internal data.** Such data is usually the result of calculations within the program.

ASSIGNMENTS The ability to create values requires some mechanism to assign values to variables. Such an assignment can take on a variety of forms in an algorithm. Some of the forms follow those of popular programming languages, while others are more English-like. Following are some examples:

```
let A = 17
set time to zero
total-cost := net-cost + tax
```

Any one of these forms, or a combination of these forms, is acceptable for representing an assignment statement. While the more English-like forms are preferable for readability, it is likely that the more algebraic forms will be necessary from time to time.

EXPRESSIONS The above results from the need to express computations, which are naturally algebraic in nature. While a form such as

```
set total-cost to net-cost plus tax
```

is understandable, it is perhaps a bit clumsy. Algebraic expressions can be formed using the standard programming symbols of +, −, *, /, ^, and ( ) for addition, subtraction, multiplication, division, exponentiation, and parentheses, respectively.

It is important that the standard method of evaluating expressions be observed. This means that parenthesized expressions are evaluated first, followed by exponentiation, then multiplication and division, and finally addition and subtraction. This **order of precedence** is summarized in the table shown in Figure 2.3-1. The operations in an expression are performed from the top down in the order shown. For example, all multiplications in an expression are performed before any additions or subtractions. For operators on the same level, such as multiplication and division, the operations are performed left to right. This is true for addition and subtraction as well.

( ), −	parenthesis, unary minus
^	exponentiation
*, /	multiplication, division
+, −	addition, subtraction

FIGURE 2.3-1: A precedence chart for standard mathematical operations. Precedence increases going up the chart. A higher-precedent operation is executed before one of lower-precedence. Thus, multiplication is performed before any additions, unless the additions are in parentheses.

For example, the expression

A * B + C * D

is evaluated by first calculating A * B, then C * B, and finally adding the two previous calculations. In a fully parenthesized form, this would look like:

(A * B) + (C * D)

If we had wanted to cause B to be added to C before any multiplications were done, we could have used parentheses to change the order of evaluations:

A * (B + C) * D

In this case, B is added to C first. Then, this result is multiplied by A (because we go left to right). This next result is finally multiplied by D.

External Data

There will also be a need to express an ability to obtain data from outside the program. This **external data** will usually come from either a user or a file. There are a number of words that can be used to express both input and output operations. It is not generally important which terms you use, but some consistency is helpful in keeping the actions unambiguous.

INPUT In the case of input, terms such as *read, input,* and *get* can be used. It is a good idea to have different terms for inputting done from various sources, such as from a user as opposed to from a file.

This makes it easier to know what action is being referred to when reading the algorithm. I generally prefer to use the term *read* for inputting from a file, and *get* for receiving input from a user. For example:

```
get the date
read a record from file A
```

The generic term *input* might be used to indicate *all* types of input. A phrase identifying the source of the input would then be required, such as in:

```
input the data from user
input a record from file A
```

OUTPUT Again, there are different possibilities for terms to represent outputting data to files and users. In addition, there may be more than one way to output a value to a user. For instance, it may make sense to distinguish between values that are output to the monitor screen and those sent to a printer device. Having special keywords to represent each of these situations would make the algorithm more readable. It should be noted, however, that there is no requirement to make such distinctions, and that they should only be made if you feel they would help make the algorithm more understandable.

Following are some examples:

```
output the record to the employee file
display the date
print the report
```

In these examples, the term *output* is used to refer to files, *display* indicates data that is listed on the user's monitor screen, and *print* sends the data to a printer. For any other devices, the generic term *output* can be applied. In addition, it may strike you as simpler to just say

```
output the date to the user
output the report to the printer
```

Here, the single term *output* is used for all devices. Additional information is included in the statement to indicate the destination of the output. This is perhaps more readable than the other forms given above.

References to Other Algorithms

In the first example given in this section, we saw how another module's algorithm could be referenced simply by using the name of the module. Wherever the name is used, we can think of making a subroutine call, suspending the current module until the called routine has been completely executed. Execution of the suspended module would be resumed following the completion of the called module.

Another aspect of referencing modules is the data upon which the called module operates. Since the purpose of calling another module is to have some specific function performed, it is reasonable to assume that there is some relationship between the results expected of the **calling** module, and those expected of the **called** module. Indeed, it is usually the case that the results of the calling module are **dependent** upon the results of the called module.

Consider briefly the algorithm given in Figure 2.3-2 to calculate an average for a list of numbers. In this algorithm, the sub-module *Calculate-Total* is referenced. Within the sub-module, a FOR loop is performed *number-of-items-in-list* times. Where does this module get

FIGURE 2.3-2: An algorithm to calculate an average for a list of numbers.

```
Calculate-Average.
BEGIN
  Get-List;
  Calculate-Total;
  set average to total-list / number-of-items-in-list;
  output average to user
END;

Get-List.
BEGIN
  get number-of-items-in-list from user;
  FOR I := 1 to number-of-items-in-list DO
    get item I of list from user
END;

Calculate-Total.
BEGIN
  set total-list to zero;
  FOR I := 1 TO number-of-items-in-list DO
    add item I of list to total-list
END;
```

this value? Obviously, the module *Get-List* is the original source of the value for this variable. However, should we assume that every value known in one module is also known in every other module? Such values are called **global** values. A similar situation exists with *total-list* and with the entire list of values being averaged.

In algorithms, we typically try to avoid the details about how certain things are going to be accomplished. For instance, we did not specify how input values are actually going to be obtained. This is a detail best left to the implementation phase.

We could take the same approach with where variables get their values. However, there is some reason to suspect that introducing a simple mechanism for **passing** data back and forth between modules would be a good idea. Such a mechanism should eliminate another potential source for misunderstanding later on. In addition, having this detail already dealt with in the algorithm will make the implementation phase a little smoother.

The mechanism of interest is called **parameter passing.** A parameter is simply the name of a variable that will be shared between modules. We can differentiate between an **input parameter,** which is a variable whose value is passed *to* a module, and an **output parameter,** a variable whose value is passed *from* a module. An **input/output parameter** has the dual role of both the input parameter and the output parameter.

The algorithm in Figure 2.3-3 is a rewrite of the example given above for calculating the average of a list of numbers, using the parameter scheme. For the *Get-List* module, all the parameters are output parameters. For the *Calculate-Total* module, *list* and *number-of-items-in-list* are input parameters, while *total-list* is an output parameter.

Using such parameter specifications makes it more obvious exactly what values the called module will be using from elsewhere in the program, and what values are expected in return.

Some Examples

We will now look at a few examples of algorithms that perform some fairly common functions. These will be in the form of modules that might be called from some other module, and parameter lists will be used to let us know that the values of certain variables will be obtained from outside the module.

```
Calculate-Average.
BEGIN
  Get-List(list,number-of-items-in-list);
  Calculate-Total(list,number-of-items-in-list,total-list);
  average := total-list / number-of-items-in-list;
  output average to user
END;

Get-List(list,number-of-items-in-list).
BEGIN
  get number-of-items-in-list from user;
  FOR I := 1 TO number-of-items-in-list DO
    get item I of list from user
END;

Calculate-Total(list,number-of-items-in-list,total-list).
BEGIN
  set total-list to zero;
  FOR I := 1 to number-of-items-in-list DO
    add item I of list to total-list
END;
```

FIGURE 2.3-3: An algorithm to calculate the average of a list of numbers, using parameter lists to pass information to other algorithms.

Exchanging Elements in a List

This first algorithm is probably familiar to you. Its purpose is to exchange the values of two elements of a list of values. This function has a number of uses, and we will later build upon this simple module to create a very useful tool. A first attempt at such a module might be:

```
Exchange(x,y).
BEGIN
  exchange the x^th element of the list with the y^th element
    of the list
END;
```

The problem is that we already knew this much before. We must further specify what the term *exchange* means in this context. We might try:

```
Exchange(x,y).
BEGIN
   set the x^th element of the list = the y^th element
   set the y^th element of the list = the x^th element
END;
```

```
Exchange(x,y,list).
BEGIN
  temporary := element at x^th position in list;
  element at x^th position := element at y^th position in list;
  element at y^th position := temporary
END;
```

FIGURE 2.3-4: An algorithm for exchanging the values of two numbers in a list.

Unfortunately, this will result in both elements getting the value of the y^th element, since the first instruction wiped out the original value of the x^th element.

Figure 2.3-4 gives the correct algorithm for this function. It was necessary to introduce a variable which will temporarily hold the original value of the x^th element of the list. In this way, that value is preserved for placing into the y^th position.

Find the Largest Value in a List

Another algorithm dealing with lists is one that will return a pointer to the largest value in a list. The **linear search** technique demonstrated in this algorithm is quite commonly used.

First, let's imagine how we, as human beings, would perform this function. Consider the following list of values:

17 34 16 21 66 13 17 0 62

How do we find the largest number? We can easily pick out that 66, in this case, is the largest, but *how* did we select it? The method involves quickly scanning the list until we recognize the largest. But this still doesn't give us an acceptable method that might be explained to someone else.

Imagine that you can't easily scan the entire list with your eyes. Perhaps the list is too long, or the numbers are hidden from view. For instance, imagine drawing the numbers out of a hat one by one. In such a case, you might look at the first number of the list and, for the present, assume that it is going to be the largest number in the list. This may later be disproved, but for now it is sufficient. For the list given above, the first number is 17.

The next step is to compare the next number drawn from the hat to 17. Since 34 is clearly larger than 17, we don't need to be concerned with 17 anymore. Thirty-four is now the largest number in the list so far.

We would next draw the third number, 16, and compare it to the current largest number, 34. Since the new number is not larger than the current largest number, we can ignore the new number. If at this point we had looked at all the numbers in the list, we would have found the largest number, namely 34. But since we have more numbers, we must continue comparing each new number to the current largest number.

Figure 2.3-5 gives an algorithm that performs such a search. The algorithm must have the list and the number of values in that list made available to it as input parameters. It will return a pointer (i.e., the position in the list) to the largest value in the list as an output parameter.

Selection Sort

Finally, we can combine the previous two algorithms to sort a list of values in ascending order. This is done by finding the position of the largest value in the list and exchanging it with the last element in the list. This places the largest value at the very end, in the position that it would properly occupy if the list were already sorted. Next, we select the second largest number in the list by looking at all the values except the last one. Once this second largest number is found, it is exchanged with the next-to-last element in the list, placing the second largest number just before the largest number in the list.

These steps are performed iteratively until all of the elements of the list have been placed in their proper places. The final result is a sorted list. Figure 2.3-6 gives the final algorithm for this selection

FIGURE 2.3-5: An algorithm for a routine to find the largest value in a list. It returns a pointer to the position of the largest value.

```
Find-Largest(list,number-of-items-in-list,position-of-largest).
BEGIN
  current-largest := 1;
  FOR I := 1 TO number-of-items-in-list DO
    IF element at Ith position in list > element at current-largest position
      THEN current-largest := I;
  position-of-largest := current-largest
END;
```

```
Selection-Sort(list,number-of-items-in-list).
BEGIN
  FOR J := number-of-items-in-list DOWNTO 1 DO
  BEGIN
    Find-Largest(list,J,position-of-largest);
    Exchange(J,position-of-largest)
  END
END;
```

FIGURE 2.3-6: An algorithm for a routine that performs a selection sort, using the previously defined Find-Largest and Exchange routines.

sort. It takes the list and the number of values in the list as input parameters. The sorted list is returned as an output parameter. Try using this algorithm on the list of numbers given above.

A Complete Program Algorithm

We could combine the algorithm we defined previously for inputting the list of values with the *Selection-Sort* algorithm to form an algorithm that will function as a complete program to read in and sort a list of numbers. This is shown in Figure 2.3-7.

Some Thoughts About Algorithm Development

In the examples presented so far, the algorithms were really developed "out of thin air," i.e., there was no analysis phase to produce preliminary design tools such as data definitions or HIPO charts.

FIGURE 2.3-7: An algorithm for a complete program that performs a selection sort. The Get-List algorithm is essentially the same as the one described in Figure 2.3-2.

```
Sort-a-List.
BEGIN
  Get-List(list, number-of-items-in-list);
  Selection-Sort(list, number-of-items-in-list);
  Output-the-List(list, number-of-items-in-list)
END;

Output-the-List(list, number-of-items-in-list).
BEGIN
  FOR I := 1 to number-of-items-in-list DO
    output element in I^th position in list to user
END;
```

This obviously will not be the case when you are developing a real application.

These additional pieces of information can be used to help develop the logic of the algorithm. The HIPO charts tell you what modules need to be developed. Their specifications give a general description of the function of each module. The data definitions provide the basis for the variable dictionary of each module.

With these tools in hand, an effective way to begin the algorithm development is to describe what the program's ultimate output is expected to be. This provides a target on which to focus while developing the details of the logic. Envision what you want the output of the program to look like, then try to transpose this vision to paper, laying out, in general, where you want things to appear. Treat a single page of paper as one screen that will be displayed to the user.

Once you have an idea about the expected outcome, begin to define the algorithm with the highest level module (the main program). Define what input values this module will need in order to produce the final output of the program. Also, define the output to be generated by this module. You must devise the statements that will take the inputs and transform them into the expected outputs.

This technique can be followed in a top-down, a bottom-up, or a combination fashion.

A Final Example

Figure 2.3-8 gives an H-chart for the *Salesperson-Estimate* module of the carpet store problem. In this example, we will develop the algorithms for this module and its sub-modules.

We can begin developing the algorithms in either a top-down manner or a bottom-up manner. The top-down approach would require us to develop the algorithm for the *Salesperson-Estimate* module first, then develop the algorithms for the four sub-modules on the next level down, and finally, the *Net, Discounts,* and *Total* sub-sub-modules.

A bottom-up approach could begin in several places. It could begin with any of the *Enter-Customer-Information, Enter-Room-Information,* or *Output-Report* modules, since they are terminal modules, i.e., they have no sub-modules. Other places to start would be any of the sub-modules below the *Calculate-Costs* module.

I prefer using the bottom-up approach. It is a little simpler and prevents additions to algorithms already developed using the top-down approach, such as when a detail is encountered in a lower-

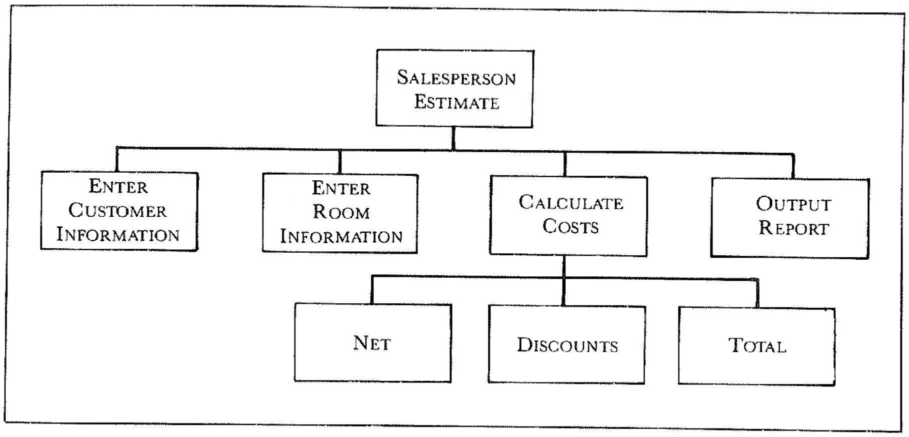


FIGURE 2.3-8: An H-chart for the *Salesperson-Estimate* function of the carpet store application.

level module that should have been handled in a higher-level module. In bottom-up development, I know all of the details of the modules underneath the one I am working on. This makes it easier to spot missing details. In addition, it lets troublesome details be "percolated" up to a more global module. Since such details tend to be global in nature, this makes more sense. In top-down development, I'm never quite sure where certain details will be taken care of, since the lower-level modules have not yet been defined in sufficient detail.

Let's begin developing the algorithms for this application with the *Enter-Customer-Information* module. This is a logical place to start, since it begins to define the set of inputs that must be collected from the user. We could also have begun with the *Output-Report* module, since it defines the outputs that the program must generate.

Figure 1.2-2 on page 24 gives the form that the program must create for the salesperson estimate. We can use this as a target for determining what inputs need to be entered by the user, what calculations need to be performed, and what the final outputs will be.

Before beginning to write the algorithm, we should prepare a variable dictionary. This dictionary is based on the data dictionary developed during the preliminary design. It also should include a brief description of the purpose of each variable. These variables will be used in the algorithms. It will undoubtedly be necessary to add variables to the dictionary as the algorithms are further developed. Figure 2.3-9 shows the variable dictionary for the algorithms that follow.

bill-name:	the name of the individual or company that is to be billed; (billing-name).
bill-address:	the address for the person or company specified by bill-name; (billing-address).
co-contact:	the name of the contact person if bill-name is a company; (contact-name).
phone:	the phone number of the bill-name or company contact; (phone).
date:	today's date; (date).
num-rooms:	the number of rooms to be carpeted; integer, range 1. .7, inclusive.
room[1. .7,1. .3]:	an array that holds the room information. Each row is the information for one room. The individual information is specified below:
room[i,1]:	the width of the I^{th} room; (width).
room[i,2]:	the length of the I^{th} room; (length).
room[i,3]:	the area of the room in square yards; (square-yards).
room-prices[1. .7,1. .3]:	an array that holds the price information for each room. Each row is the information for one room. The individual information is specified below:
room-prices[i,1]:	the cost per square yard of the carpet selected for the I^{th} room; (square-yards-price).
room-prices[i,2]:	the cost per square yard for padding of the I^{th} room, if it is selected by the customer; (padding-price).
room-prices[i,3]:	the cost per square yard for installation of the I^{th} room. if selected by the customer; (installation-price).
room-cost[1. .7]:	the total cost per square yard for each room, including the carpet, padding, and installation costs; (room-total).
net-amt:	the net cost of all the rooms, not including discounts or tax; (total);
width:	the width of a room; (width).
length:	the length of a room; (length).
carpet-code:	a code that uniquely identifies a style of carpet in the store; (carpet-code);
carpet-price:	the per-square-yard price of a particular carpet; (square-yards-price).
padding-price:	the cost per square yard for padding; this is a constant for all padding, value 2.95.
install-price:	the cost per square yard for installation; this is a constant for any installation; value 3.95.
customer-file:	the file containing information about a customer's year-to-date purchases. This information is used to determine if a customer is a "regular" customer. (customer-file).
customer:	a record from the customer file; (customer).
room-code[1. .7]:	the code of the carpet selected for each of the rooms. (carpet-code).
ytd-purchases:	the year-to-date purchases for a particular customer; (year-to-date-purchases).
dscnt-percent:	the amount of the discount the customer is entitled to; value range of 0, 0.05, 0.07, 0.10, 0.12, 0.15.
discount-amt:	the dollar amount of the discount a customer is entitled to; (total).
subtotal:	the net amount minus the discount amount; (total).
tax-rate:	the percent sales tax applied; constant, value 0.06.
tax:	the dollar amount of the tax; (total).
salesperson:	the name of the salesperson entering the order; (salesperson).

FIGURE 2.3-9: A variable dictionary for the algorithms given in Figures 2.3-10 to 14. The items in parentheses at the end of the definitions refer to the original data definitions given (see Figures 1.4-1 through 1.4-4). Note that some of the items (such as the padding price and tax rate) were not specified in the original data definitions, and so have been more fully specified in this variable dictionary. This is because these items were not part of any files, which is what the data definitions currently represent. The more formal approach would be to add these items to the data dictionary, then refer to them in the variable dictionary.

```
Enter-Customer-Information.
BEGIN
  REPEAT
    INPUT date FROM USER;
    INPUT bill-name FROM USER;
    INPUT bill-address FROM USER;
    INPUT co-contact FROM USER;
    INPUT phone FROM USER;
    OUTPUT date, bill-name, bill-address, co-contact, phone TO USER
  UNTIL information is correct
END;
```

FIGURE 2.3-10: The algorithm for the *Enter-Customer-Information* module of the carpet store application.

Figure 2.3-10 gives the algorithm for the *Enter-Customer-Information* module.

Figure 2.3-11 gives the algorithms for the *Enter-Room-Information* module. Since this module introduces additional variables, the variable dictionary needs to be updated.

In addition, note that there is a reference to a module—the *Find-Price* module—that was not defined in the H-chart. During the development of the *Enter-Room-Information* module, it was discovered that there was no defined mechanism for determining the price of a particular carpet. It would be most convenient to have a file that contains such information about every carpet the store sells. Each carpet has a unique carpet code associated with it. This code can be used to locate a particular carpet's price in the file.

Using such a file limits the errors that might occur on pricing, and allows the management to easily change a carpet's price. Introducing such a file would probably require adding to the data dictionary.

The MOD operator used in this algorithm produces the remainder of dividing the number on the left from the number on the right. The method shown rounds up the width and length of the room to the next number that is evenly divisible by three.

Finally, note that some of the details of the algorithm are somewhat vague, such as "If user wants padding" or "until correct." These statements will be fleshed out when the algorithms are finally implemented in a programming language.

Next, let's develop the algorithms for the *Calculate-Cost* module, shown in Figure 2.3-12. Begin by discussing the sub-modules.

Although the calculations performed in the *Net* module could

```
Enter-Room-Information.
BEGIN
  padding-price := 2.95;
  install-price := 3.95;
  REPEAT
    INPUT num-rooms FROM USER
  UNTIL correct;
  FOR I := 1 TO num-rooms DO
  BEGIN
    REPEAT
      INPUT width, length FROM USER
    UNTIL correct;
    room[I,1] := width + 3 - (width MOD 3);
    room[I,2] := length + 3 - (length MOD 3);
    room[I,3] := (room[I,1] * room[I,2]) / 9;
    REPEAT
      INPUT carpet-code EROM USER
    UNTIL correct;
    room-code[I] := carpet-code;
    Eind-Price(carpet-code,carpet-price);
    room-prices[I,1] := carpet-price;
    IF user wants padding
    THEN
      room-prices[I,2] := padding-price
    ELSE
      room-prices[I,2] := 0;
    IF user wants installation
    THEN
      room-prices[I,3] := install-price
    ELSE
      room-prices[I,3] := 0
  END
END;

Find-Price(carpet-code,carpet-price).
BEGIN
  REPEAT
    INPUT carpet-info record FROM customer-file
  UNTIL a match with carpet-code found;
  get the carpet-price from the carpet-info record just found
END;
```

FIGURE 2.3-11: Algorithms for the *Enter-Room-Information* module of the carpet store application. Parameters were added to the Find-Price reference in order to aid readability. Note the initialization of the constants *padding-price* and *install-price* at the beginning of this module.

have been done when the room information was entered in a previous module, it would not have followed our modularity principles to do so. The inefficiency introduced, as a result, is that there are two loops that step through the room information instead of only one. In this case, however, the inefficiency is more than tolerable, and the redundancy adds to the understandability of the algorithms.

```
Calculate-Costs.
BEGIN
  Net(net-amt);
  Discounts(net-amt, dscnt-percent);
  Total(net-amt, dscnt-percent, subtotal, tax, total)
END;

Net(net-amt).
BEGIN
  net-amt := 0;
  FOR I := 1 TO num-rooms DO
  BEGIN
    room-cost[I] := (room-prices[I,1] + room-prices[I,2] +
                    room-prices[I,3]) * room[I,3];
    net-amt := net-amt + room-cost[I]
  END
END;

Discounts(net-amt, dscnt-percent).
BEGIN
  REPEAT
    INPUT next customer record FROM customer-file
  UNTIL customer's record found;
  determine if regular customer using year-to-date purchases
          from the customer record;
  IF regular customer or order is not on credit
  THEN discount-table(dscnt-percent)
END;

Discount-Table(dscnt-percent).
BEGIN
  IF regular customer
  THEN
    CASE net OF
      0. .1999:        dscnt-percent := 0.05;
      2000. .4999:     dscnt-percent := 0.10;
      5000. .9999:     dscnt-percent := 0.12;
      OTHERWISE:       dscnt-percent := 0.15
    ENDCASE
  ELSE
    CASE net OF
      0. .1999:        dscnt-percent := 0.0;
      2000. .4999:     dscnt-percent := 0.05;
      5000. .9999:     dscnt-percent := 0.07;
      OTHERWISE:       dscnt-percent :=0.10
    ENDCASE
END;

Total(net-amt, dscnt-percent, subtotal, tax, total).
BEGIN
  discount-amt := net-amt * dscnt-percent;
  subtotal := net-amt − discount-amt;
  tax := subtotal * tax-rate;
  total := subtotal + tax
END;
```

FIGURE 2.3-12: The algorithms that make up the *Calculate-Costs* module of the carpet store application.

The *Discounts* module calculates any discounts to which a customer is entitled. It uses another module called *Discount-Table,* which sets the percentage rate of discount for the customer. It was separated into its own module since this table will also be referenced in the *Orders* module. It is entirely reasonable to separate a duplicate function into its own module. In this way, only a single module must be maintained if the discount mechanism or rates change in the future.

Figure 2.3-13 gives the algorithm for the *Output-Report* routine. The term *report* has been taken somewhat loosely here to include outputting the appropriate information to the estimate file as well as to a printer for the customer. In this case, this is reasonable since the formats of the two outputs are so similar. In other applications, this might not be the case.

Figure 2.3-14 shows the algorithm for the main module. This is little more than a driver program for the other modules. Note, however, that the salesperson's name is input in this module. You may have wondered when this piece of information would be entered. Indeed, it probably became obvious that this function was required when developing one of the other modules. However, since this function did not appropriately fit into any of the other modules, it was pushed up into this higher-level module.

Figure 2.3-15 shows how the *Orders* module might be written. This is included to show the differences between this section of the system and the *Salesperson-Estimates* section. Note that the logic to determine if the customer is a "regular" customer has been included in this module. In the estimate section of the program, this logic was in the *Discounts* module. This logic is required here by the additional

FIGURE 2.3-13: The algorithm for the *Output-Report* module.

```
Output-Report.
BEGIN
  OUTPUT date, bill-name, bill-address, co-contact, phone TO
            PRINTER and Estimate File;
  OUTPUT room headings TO PRINTER;
  FOR I := 1 to num-rooms DO
    OUTPUT room information TO PRINTER and Estimate File;
  OUTPUT net, discount-amt, subtotal, tax, total TO PRINTER and
            Estimate File;
  OUTPUT salesperson TO PRINTER and Estimate File
END;
```

```
Salesperson-Estimate.
BEGIN
  REPEAT
    INPUT salesperson FROM USER;
  UNTIL correct;
  Enter-Customer-Information;
  Enter-Room-Information;
  Calculate-Costs;
  Output-Report
END;
```

FIGURE 2.3-14: The final algorithm for the *Salesperson-Estimate* module of the carpet store application. Note that its form is that of a driver routine that simply calls other detailed sub-modules.

actions that need to be done when an order is actually entered. For the *Order* section of the application, the *Discounts* module would not need to include the logic for determining if the customer is "regular." The hierarchy of these modules can be seen in Figure 1.3-2c on page 38.

Also note the inclusion of yet another file, the Pending File. We needed some way of holding onto orders awaiting credit approval. This can be handled neatly by placing them into the Pending File until credit has either been approved or rejected. It is still unclear, however, exactly what "discarding" an order entails. Also, the logic of the module as shown will attempt to output an order regardless of whether credit has been approved or rejected. It is left as an exercise to work this out in more detail, but such details can be left until the implementation. The algorithms given are more than sufficient for moving on to the next phase of the development.

Algorithm Testing

Once the algorithms have been created, we cannot immediately jump into implementing them in code without some attempt to verify that they in fact do what we think they should. This testing consists of making up some sample data that can be used while "executing" the algorithms by hand. For the carpet store customer estimate program, this would mean making up numbers for the inputs that were specified above. Using these numbers, follow the steps of each algorithm, starting with the main program.

If the algorithms seem to generate the output that you expected, then it is generally safe to continue with the next phase of the

```
Orders.
BEGIN
  Enter-Orders;
  REPEAT
    INPUT a customer record FROM the Customer-File
  UNTIL the customer's record is found;
  determine if the customer is a regular customer using year-to-date purchases from the
     customer record;
  IF purchase is not on credit
  THEN
    schedule delivery and installation
  ELSE
  BEGIN
    IF regular customer
    THEN
      CASE credit OF

        approved:            BEGIN
                               remove order from Pending File;
                               add order to Order File;
                               OUTPUT delivery and installation invoice TO PRINTER
                             END;
        rejected:            BEGIN
                               remove order from Pending Eile;
                               discard order;
                               OUTPUT rejection report TO PRINTER
                             END;
        OTHERWISE:           BEGIN
                               schedule delivery and installation;
                               add order to Pending File;
                               OUTPUT credit request TO PRINTER
                             END
      ENDCASE
                                {end of IF regular customer, too}
    ELSE                        {not a regular customer}
      CASE credit OF
        approved:            BEGIN
                               remove order from Pending File;
                               add order to Order Eile;
                               schedule delivery and installation;
                               OUTPUT invoice TO PRINTER
                             END;
        rejected:            BEGIN
                               remove order from Pending File;
                               discard order
                             END;
        OTHERWISE:           BEGIN
                               add order to Pending File;
                               OUTPUT credit request TO PRINTER
                             END
      ENDCASE
                             {end of ELSE for IF regular customer, too}
    END;                     {end of ELSE for IF not on credit}
  Recalculate-Costs;
  Output-Report
END;
```

FIGURE 2.3-15: An algorithm for the *Enter-Order* module of the carpet store application. The items in the brackets ({ }) are comments that help explain the actions in the algorithm without affecting how the algorithm works. Such statements are included to help the readability of the more difficult passages of logic. Comments can be added anywhere in an algorithm that requires a little additional explanation.

development, i.e., implementation. If something seems to go wrong while you are tracing the algorithms' steps and the wrong output is produced, you have one or more bugs in your algorithms. These must be corrected before the implementation phase can begin. It is, perhaps, more difficult to locate bugs at the algorithm stage than at a later stage, but bugs discovered at this stage are far easier and less costly to correct than if we wait until the algorithms have been coded.

Testing and debugging techniques are discussed in detail in Chapter 5.

2.4 THE PROPER USE OF FLOWCHARTS

We have already seen how using one form of graphical representation of a program system can be an aid in designing the system. The Visual Table of Contents (VTOC) discussed in Chapter 1 allows us to quickly see the relationships of the various modules defined. A visual representation of information can often lead to quicker understanding of a concept than a verbal one.

Flowcharts are an old graphical form for representing program logic. They have fallen into great disrepute these days, partly because of their age. Flowcharts were developed back in the dark ages of programming, i.e., the 1950s. Part of the impetus for their development was undoubtedly the unstructured nature of programming languages of the day, namely FORTRAN, COBOL, and assembly language. Because of the uncontrolled use of GOTOs in the code, the program logic was often anything but clear, and a programmer had little hope of following anyone else's code, let alone his own. Flowcharts made it easier to follow the flow of the logic, since the diagrams showed not only the individual statements of the program, but also used flow lines to show the order of execution.

Many computer scientists believe that the flowchart has outlived its usefulness. They feel that, since we have moved on to using more modern programming languages that all but eliminate the GOTO, there is no need to use a technique that was developed to overcome the problems introduced by the GOTO. In addition, they figure that since modern programmers would undoubtedly be developing their programs in pseudo-code and then in a structured language such as Pascal, the code would automatically be more understandable than before, and therefore the extra step of drawing the flowcharts wouldn't be needed. Finally, it is argued that one obviously shouldn't use an inherently unstructured technique such as flowcharts in this enlightened age, since the goal is a structured program.

The problem with this analysis is that it is only half right and has several holes in it. The first hole is the assumption that everyone will be writing programs in one of the modern (i.e., structured) languages such as Pascal or the newcomer, Ada. Certainly in the microcomputer field, the most popular language will continue to be BASIC for a long time to come. In addition, much work is still most appropriately done in assembly language. Also, FORTRAN and COBOL are still powerful forces in the field. This may all change in the near future, but for now there is no ignoring these truths.

The second hole is the idea that flowcharts are unnecessary when using a structured language. Flowcharts can give a quick reference to the overall logic of a program or system that is difficult to obtain from just a program listing or an algorithm written in pseudo-code, and often help the programmer to find logic errors in a program being translated from pseudo-code to flowchart form. Any technique which makes it more likely to catch bugs is definitely worthwhile.

Finally, the art of flowcharting has been adapted to the more modern age of structuring, to the point where there is talk of "structured flowcharts." The idea is that a few simple rules can transform flowcharts into a tool that not only follows the philosophies of structured programming, but also forms a natural bridge between algorithms developed in pseudo-code and an unstructured language such as BASIC, FORTRAN, COBOL, or assembly language. In this way, flowcharts are *necessary* when programming in one of these languages.

The biggest standing problem with flowcharts, however, is that the technique is too nonstandard, or at least is too complex for programmers to take the trouble of keeping their flowcharts standard. A typical flowcharting **template** has more than twenty symbols. While it is a nice idea to give every conceivable process its own symbol so that the shape of the symbol identifies the process, some processes are poorly understood by many programmers. In addition, some companies define symbols differently. Finally, there are simply too many symbols to remember. As a result, instead of aiding in understanding, flowcharts often obscure matters.

The way symbols are used in presenting the logic of a program is also nonstandard. Many programmers simply fill in the various boxes with code! This defeats the whole purpose of using flowcharts. However, most programmers consider flowcharts to be good only for documentation purposes. They never think of using flowcharts as a design tool, and would much rather forget the design phase altogether. They would prefer instead to begin coding as soon as an idea for a piece of the program strikes them. To make matters worse, many authors promote this form of flowcharting, especially in books on BASIC. What is missed by these so-called experts is that flowcharts are indeed part of the design phase, not the implementation phase, and therefore should be language independent. As in presenting the algorithm in pseudo-code, it should be possible to implement a program in any language from a flowchart.

The purpose of this section is to present a more appropriate approach to flowcharting. First, we will define a flowchart as *a graphical representation of an algorithm.* This eliminates the problem of wanting to present actual code in the various symbols. And since we have already defined that algorithms should be developed in pseudo-code, it is easy to see that flowcharts will obviously be based upon a structured design and will therefore be structured themselves. The fact that the ultimate implementation might be in an unstructured language such as BASIC makes the flowchart indispensable. As we will see when implementation issues are addressed (Chapter 3), the flow lines within the diagram make it easier to deal with implementing the structured concepts of the algorithm in a GOTO-oriented language.

Flowchart Symbols

Rather than use the overly rich set of symbols available, we will define a small subset of symbols to use in flowcharts. Figure 2.4-1 shows the symbols we will use for the flowcharts in this book. The symbols represent all the functions necessary for presenting any algorithm. We have eliminated multiple types of the same function. For instance, in the original flowchart template, there are several symbols devoted to various ways of indicating input to or output from the logic.

We will use the single symbol of the parallelogram to show all types of I/O. When this *I/O* symbol is used in a diagram, there will be exactly one line flowing into it, and one line flowing out of it. This is in keeping with the "one in–one out" philosophy of structured programming.

The *terminal* symbol is used to indicate the beginning or end of any flowchart. This symbol will contain a name when we wish to indicate the beginning of a flowchart. There will only be one place to begin in every flowchart, again in adherence to our "one in–one

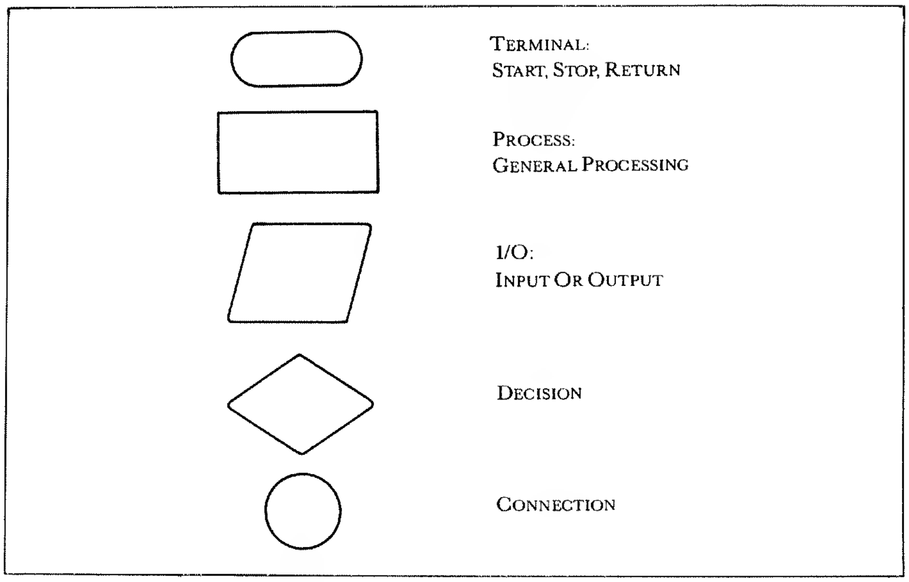


FIGURE 2.4-1: The fundamental set of flowchart symbols.

out" structured programming concept. Likewise, there will be a single exit from a flowchart, using the same symbol but containing the word "stop" or "return," depending on whether the flowchart is for a main program or a subroutine. There will be only one line coming out of or going into this box in a diagram.

The *process* box is used to indicate any processing to be done that is not represented by another symbol. This will typically be the English statements presented in the algorithm. The keywords of the algorithm are themselves represented by the diagram symbols and flow lines and so are never actually presented in the flowchart. There should be exactly one line flowing into and one line flowing out of this symbol whenever it is used in a diagram.

The *decision* symbol is the graphical representation of a binary condition. There will be one line coming into this symbol, but two leaving it, representing the two possible results (true and false) of the condition. This is not in violation of the "one in–one out" rule, but is instead a special case.

The final symbol to be used is the *connector*. The small circle is used to indicate a connection between one piece of the same flowchart and another. It is necessary because flowcharts have a habit of taking up a lot of space. Rather than trying to cram all the symbols onto a single page, and in order to avoid crossing flow lines in the diagram, it is usually a good idea to spread the symbols out. The connector is then used to link together pieces of the diagram that may have been separated on different pages. In addition, try to keep the diagram as clean and uncluttered as possible by using connectors whenever it would be necessary for two flow lines to cross.

Connectors always come in sets, with any number of connectors indicating where you are leaving from, but only a single connector for the destination, indicating where the flow is going to be continued. As a result, each connector contains a unique "address" of where to go next. This address can be as simple as consecutively numbering each connector that acts as a destination, then using that number in any connector flowing to that destination. However, in a complex diagram that covers several pages, it is often difficult to follow such a scheme.

I prefer to use a coding system that helps locate what page a destination connector might be on. Very simply, each new page gets its own number. Then, I "number" every destination connector on a given page with the number of the page and letters of the alphabet. For instance, the first destination connector on page two would be labeled "2A," the second connector "2B," etc. The first destination connector on page three would be called "3A." I use this scheme even when I know that the destination is on the same page as the point of departure. Incidentally, I have never found a need for more than three or four destination connectors on any single page.

Flowcharts for Pseudo-code

Since flowcharts will be a graphical representation of an algorithm, the easiest way to proceed is to redefine the standard pseudo-code control structures in a flowchart form. We will define the natural flow through these diagrams as being straight down or to the right, unless redirected by arrows on the flow lines. In addition, flow lines occasionally intersect. This is done instead of drawing multiple lines into a single box. It also makes the chart easier to change when adding or deleting boxes. Where two or more lines intersect, arrows will be used explicitly to indicate the proper flow. Finally, flow lines will never be allowed to cross without merging; connectors will be used to avoid this where necessary.

Sequence

Figure 2.4-2 shows how a sequence of statements would be presented in a diagram, along with the portion of pseudo-code on which the

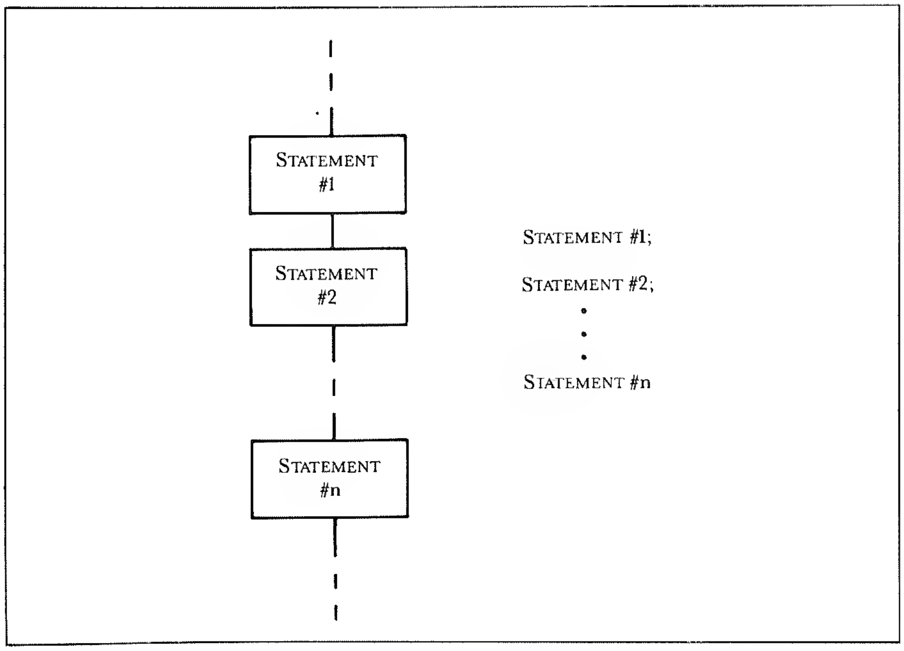


FIGURE 2.4-2: The flowchart representation of a sequence of statements.

diagram is based. Flow moves from statement #1 to statement #2 . . . to statement #n, just as it would in the pseudo-code. This type of statement presentation can occur anywhere in a diagram, just as a sequence of statements can occur anywhere in a piece of pseudo-code. Placing a BEGIN and END block around several statements would not change the way the statements are presented in the flowchart. There will be other indications that the statements make up a block or compound statement.

Selection

The IF statement is one of the pseudo-code constructs most often abused in the development of flowcharts. Instead of being used simply to indicate alternatives of statements, the decision symbol—the graphical representation of the IF construct—is often used to indicate a branch in the program logic, much like a fork in a road. This unfortunately leads to a highly unstructured program when it is finally coded.

Figure 2.4-3 shows a proper flowchart for an IF-THEN-ELSE

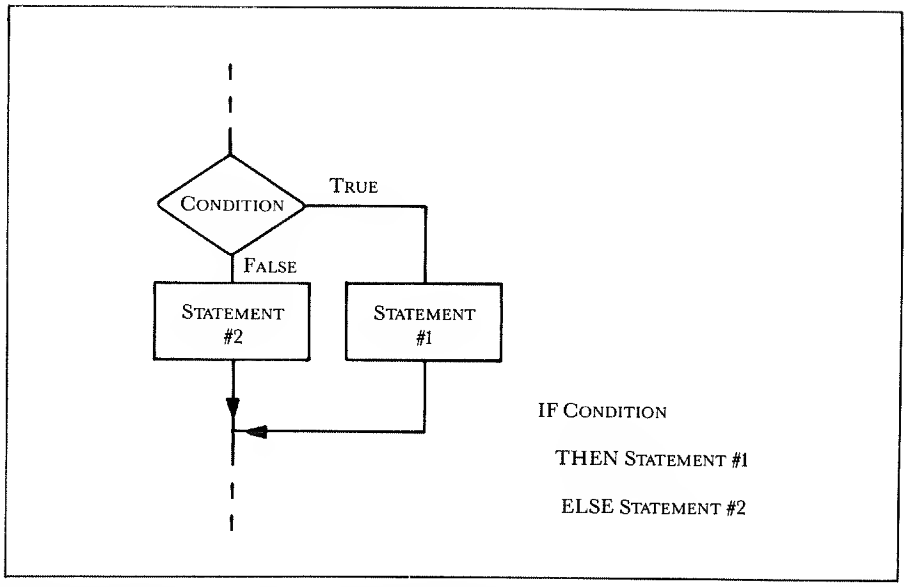


FIGURE 2.4-3a: Figure (a) shows a style for representing the IF-THEN-ELSE statement. Figure (b) shows how a dashed box can be used when the statement for the decision condition won't fit into the diamond-shaped box.

construct (a). The condition specified in the decision symbol is the same as the condition that was presented in the pseudo-code. Note that the flow enters the top of the decision symbol and then takes one of two alternative paths, depending upon the outcome of the decision. Also note that the flow merges again after the two alternative paths have been completely specified. This is because flow will be passed on to whatever statement follows the IF-THEN-ELSE.

As in the sequence construct above, having a compound statement in either or both of the alternative paths is no real problem. It simply means that there will be more than one process box along a given path. Again, the BEGIN and END keywords don't have any specific symbols to represent them in the flowchart. Any block of statements appears simply as a sequence in the appropriate place in the diagram.

It is often difficult to fit an entire condition or statement within the small flowchart symbols. In such a case it is helpful to have a way of extending the symbol so that more detailed descriptions can be used. Figure 2.4-3 shows how this can be done by using a dashed box (b) to present comments or a more detailed specification of some

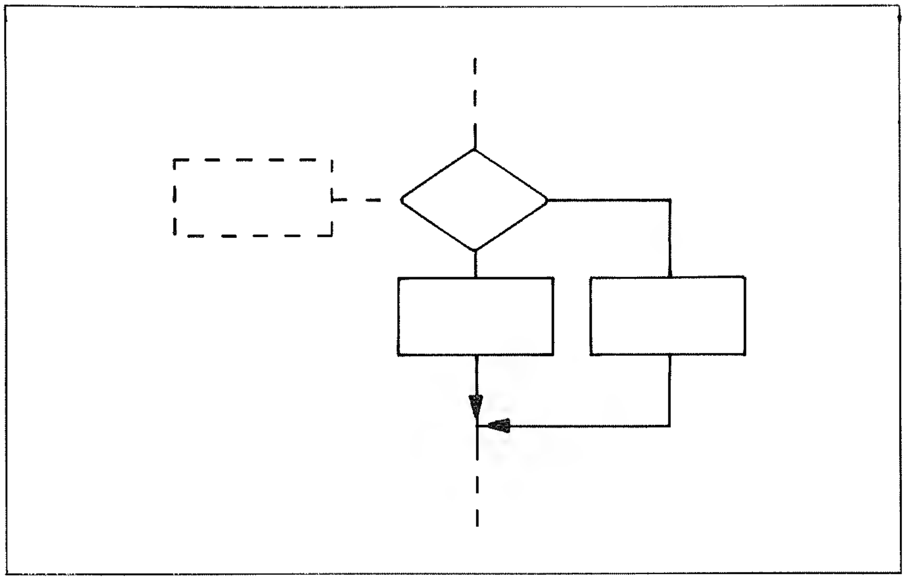


FIGURE 2.4-3b

condition. In addition, this box can be used to add comments to any type of symbol or group of symbols.

As described in section 2.2, The Use of Pseudo-code, it is often the case that the ELSE portion of an IF statement isn't required. Figure 2.4-4 shows how this can be accommodated in the flowchart, simply by having no processing of any kind specified within the *false* clause. Flow "falls through" the decision box straight on to the next statement following the IF statement.

The CASE construct is much more difficult to present neatly in a graphical form than any other construct. This is because of the numerous possible choices that can be made as to the next statement to be executed. Figure 2.4-5 shows one approach, using standard symbols. By using the more conventional symbols that have already been defined, there is some hope of actually converting this complex CASE construct into the code of a language that does not have a CASE statement.

The case selector is specified in the decision symbol, with each individual case being specified next to the flow line of the path to be followed if that case applies. Only one path will, in fact, be followed during the actual execution of the code. This diagram specifies all possible paths.

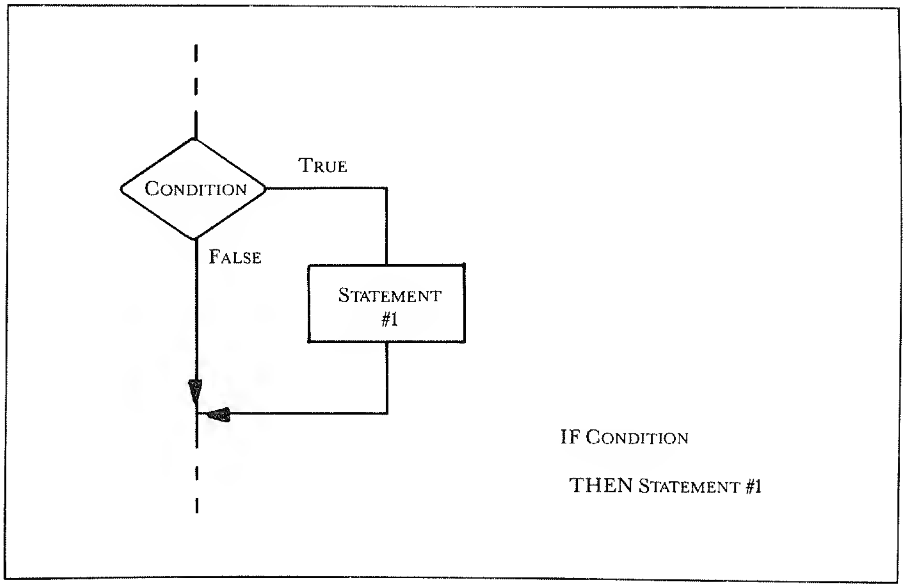
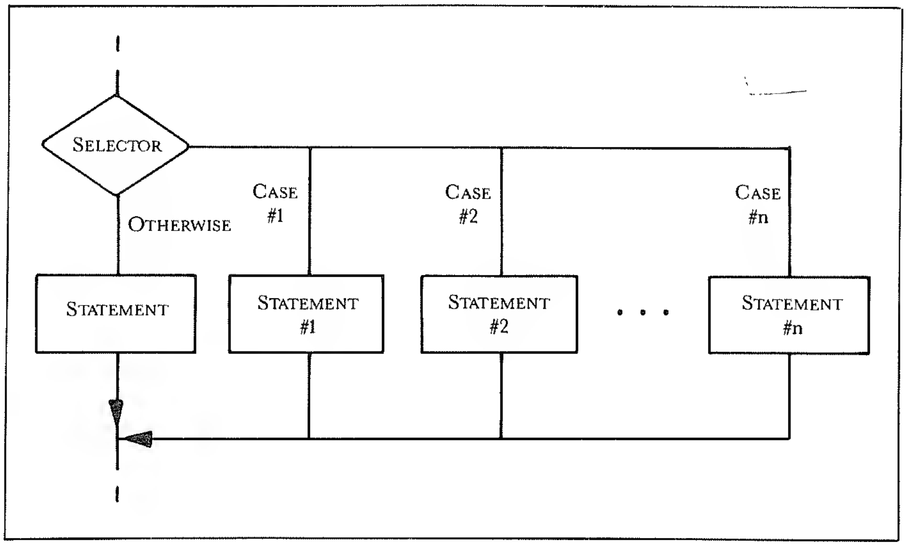


FIGURE 2.4-4: This figure shows a representation of an IF statement without an ELSE clause.

FIGURE 2.4-5: Representing the CASE statement with a flowchart. The form is easily translated into code for languages that do not contain the CASE statement.



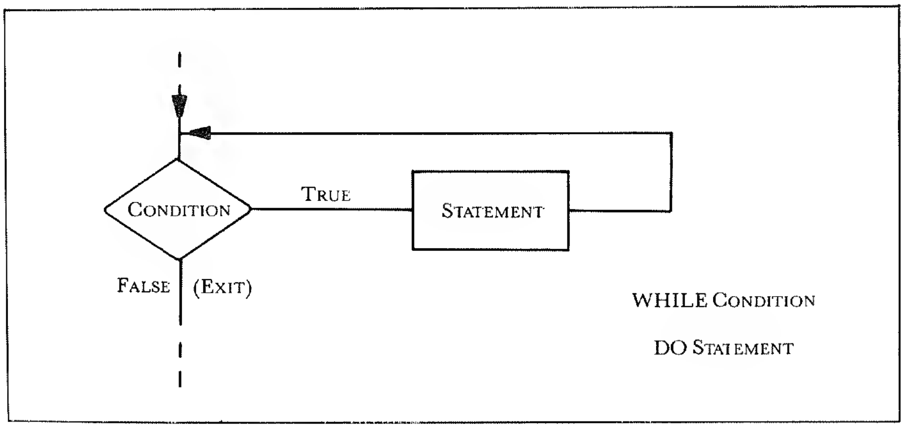
Iteration

The WHILE construct is very easily represented with a simple decision box and a loop body of statements. Figure 2.4-6 shows how this can be presented. Note that the loop flow reenters the flow line above the decision symbol, rather than drawing the line directly into the box. This is more a matter of style than necessity. Also, marking the *false* flow line out of the decision box as an exit helps to identify this series of symbols as a loop, not some weird kind of IF statement. Note too that the loop body is always performed whenever the condition is *true*. This conforms to the original definition of the WHILE loop.

Figure 2.4-7 shows the flowchart representation of the REPEAT statement. The only thing to note here is that the condition causes the loop to be repeated when it is *false*, and exits the loop when it is *true*.

The FOR type of loop presents a bit of a problem in coming up with a neat representation. The problem is that the FOR statement itself represents three distinct actions: an initialization, a condition test, and an incrementation. The flowchart in Figure 2.4-8(a) has been suggested by several authors as a clever way of representing all of these actions in a single symbol. This is simply a combination of two symbols, the process box and the decision box. The initialization of the index variable is shown in the top left portion of the box. The bottom left portion shows the incrementation of the index. The decision portion shows the test criterion for continuing the loop. Note

FIGURE 2.4-6: A representation of the WHILE loop in flowchart form.



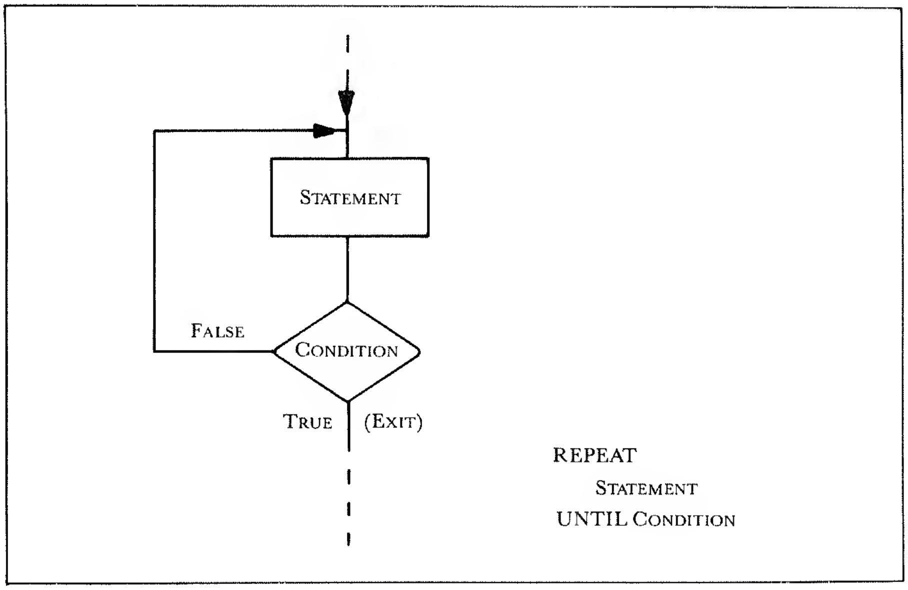
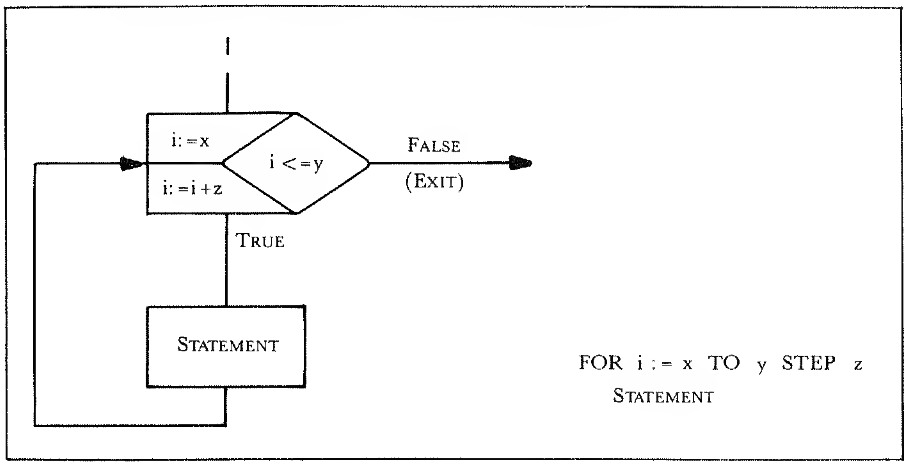


FIGURE 2.4-7: The REPEAT loop in flowchart form.

that the loop flow line does enter the side of this box, rather than entering the flow line above the box as we have seen before. This is to indicate that the initialization statement of the loop is only performed when this loop is entered for the very first time, not with each iteration of the loop.

FIGURE 2.4-8a: One method for representing the FOR loop with a flowchart.



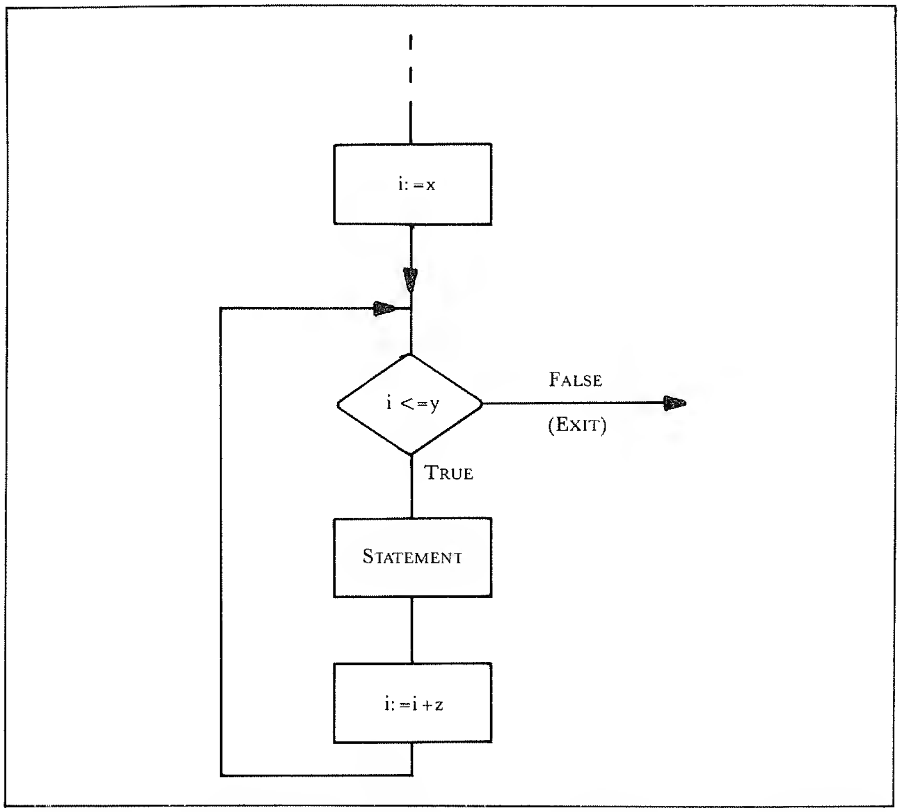


FIGURE 2.4-8b: An alternate way of representing the FOR loop in a flowchart, emphasizing the fact that this loop is based on the WHILE construct.

The only real objection I have to this format is that it is not clear from the symbols themselves whether the test is done at the top of the loop like a WHILE statement, or at the bottom like a REPEAT statement. While this small amount of ambiguity might appear acceptable, what happens when two programmers who know different dialects of BASIC attempt to use this same flowchart to implement a program for their system? They would both undoubtedly use a FOR-NEXT loop here, even though one might work like a WHILE loop, and the other like a REPEAT loop. Clearly they can't both be right.

Therefore, the only alternative to allowing this ambiguity to exist is to construct the FOR loop in flowchart form the hard way. Figure 2.4-8b shows the initialization, test, and increment for a WHILE loop implementation. Figure 2.4-8c does the same for a REPEAT loop

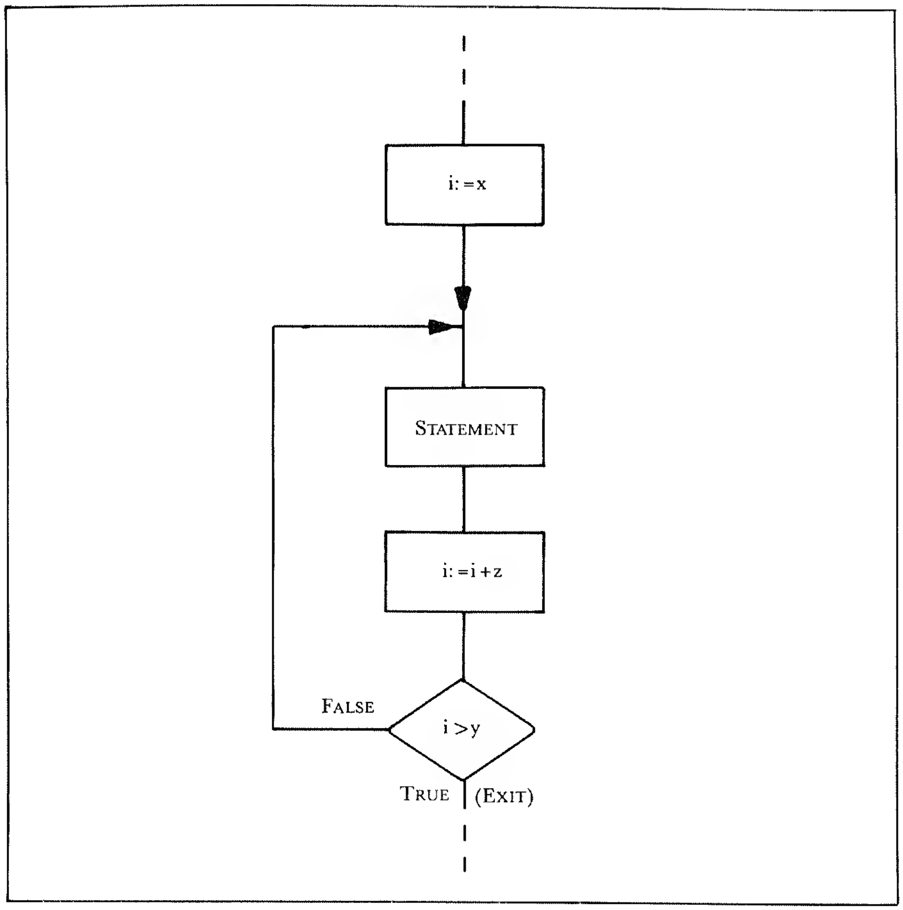


FIGURE 2.4-8c: A FOR-like loop based on the REPEAT construct.

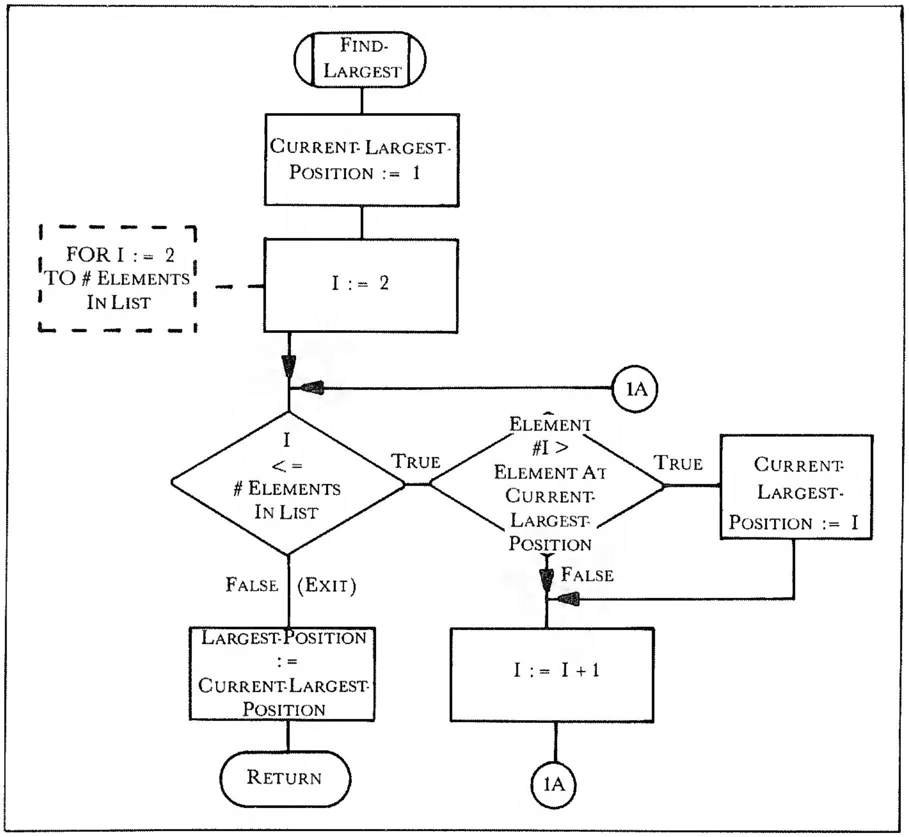
implementation. The biggest problem now is that it is not obvious from the diagram that we have a FOR loop instead of a WHILE or REPEAT loop. There is little that can be done to correct this, except to add a comment to the initialization box explaining that this is a FOR statement.

Representing References to Modules

Since we have gone to all the trouble to allow the definition of modules in our pseudo-code, we obviously need some way of representing a reference to another module within the flowchart. Each module will have its own flowchart, just as it has its own algorithm already defined. The module reference in the pseudo-code was very simple. All we had to do was write down the name of the module we wanted executed. We defined that such a reference caused the execution of the currently executing, or "calling," module to be suspended, and the "called" module execution to begin. When this called module had completed its execution, it returned control to the calling module, to the statement following the reference.

We will follow the same conventions within the flowchart for module referencing. We will simply specify the name of some other module within a process box, thus invoking and passing control to the beginning of the module referred to. In order to make the module reference more obvious, the box referring to another module is given double sides. In addition, the beginning terminal box for the called module has double sides. Finally, a module that is called will

FIGURE 2.4-9: An example of a flowchart for the *Find-the-Largest* algorithm.


FIND-LARGEST
CURRENT-LARGEST-POSITION := 1
FOR I := 2 TO # ELEMENTS IN LIST
I := 2
1A
I <= # ELEMENTS IN LIST
TRUE
ELEMENT #I > ELEMENT AT CURRENT-LARGEST-POSITION
TRUE
CURRENT-LARGEST-POSITION := I
FALSE
(EXIT)
FALSE
LARGEST-POSITION := CURRENT-LARGEST-POSITION
I := I + 1
RETURN
1A

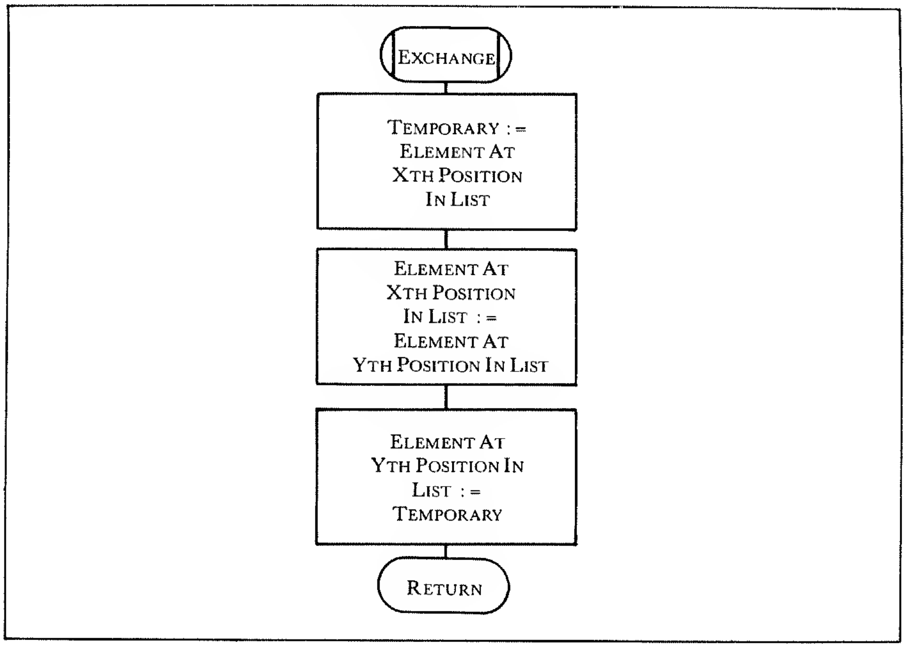
have "return" in its terminal ending symbol instead of "stop." This indicates that control passes back to the calling routine following completion of the logic for the called module.

Figure 2.4-9 shows an example of a flowchart for the *Find-the-Largest* algorithm developed in section 2.3. Note the use of a connector to complete the loop of the FOR construct. Since this routine is used as a module in the *Selection-Sort* algorithm, the beginning terminal symbol is double-sided and the ending terminal symbol says "return." Finally, the FOR loop has been identified by the use of a comment box at the initialization statement.

Figure 2.4-10 shows the flowchart for the *Exchange* algorithm. Note the use of the pseudo-variable TEMPORARY. Also notice that, even though we explicitly state the parameters that will be passed to this module in the pseudo-code, there is no mechanism in the flowchart for showing parameters.

Figure 2.4-11 shows how the two modules above are referenced from another flowchart. When we reach the *Find-Largest* box in the *Selection-Sort* flowchart, we would move to the *Find-Largest* flowchart, following it until we are led to the "return." We would then continue

FIGURE 2.4-10: A flowchart for the *Exchange* algorithm.


EXCHANGE
TEMPORARY := ELEMENT AT XTH POSITION IN LIST
ELEMENT AT XTH POSITION IN LIST := ELEMENT AT YTH POSITION IN LIST
ELEMENT AT YTH POSITION IN LIST := TEMPORARY
RETURN

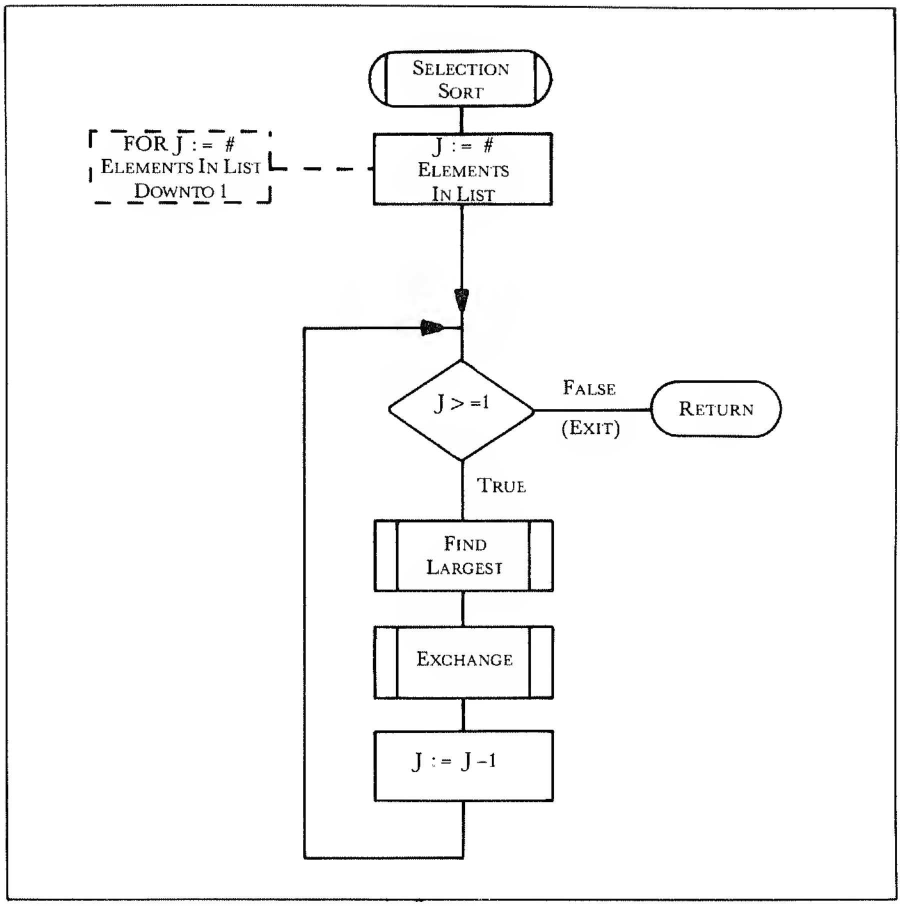


FIGURE 2.4-11: A flowchart for the *Selection-Sort* algorithm.

following the *Selection-Sort* flowchart with the box that indicates calling the *Exchange* module.

Finally, Figure 2.4-12 shows the flowchart for the *Sort-a-List* program. After the call to the *Selection-Sort* module, the execution continues with the FOR loop, which outputs the sorted list.

These flowcharts will be used to actually implement the code for this program. Note that there is very little in the way of "language specific" statements in the flowcharts. Therefore, it should be easy to implement this logic in nearly any programming language. In addition, the flowcharts provide one more level of documentation for the system. The only thing that you have to remember is that the flowcharts must be changed whenever the logic of the program

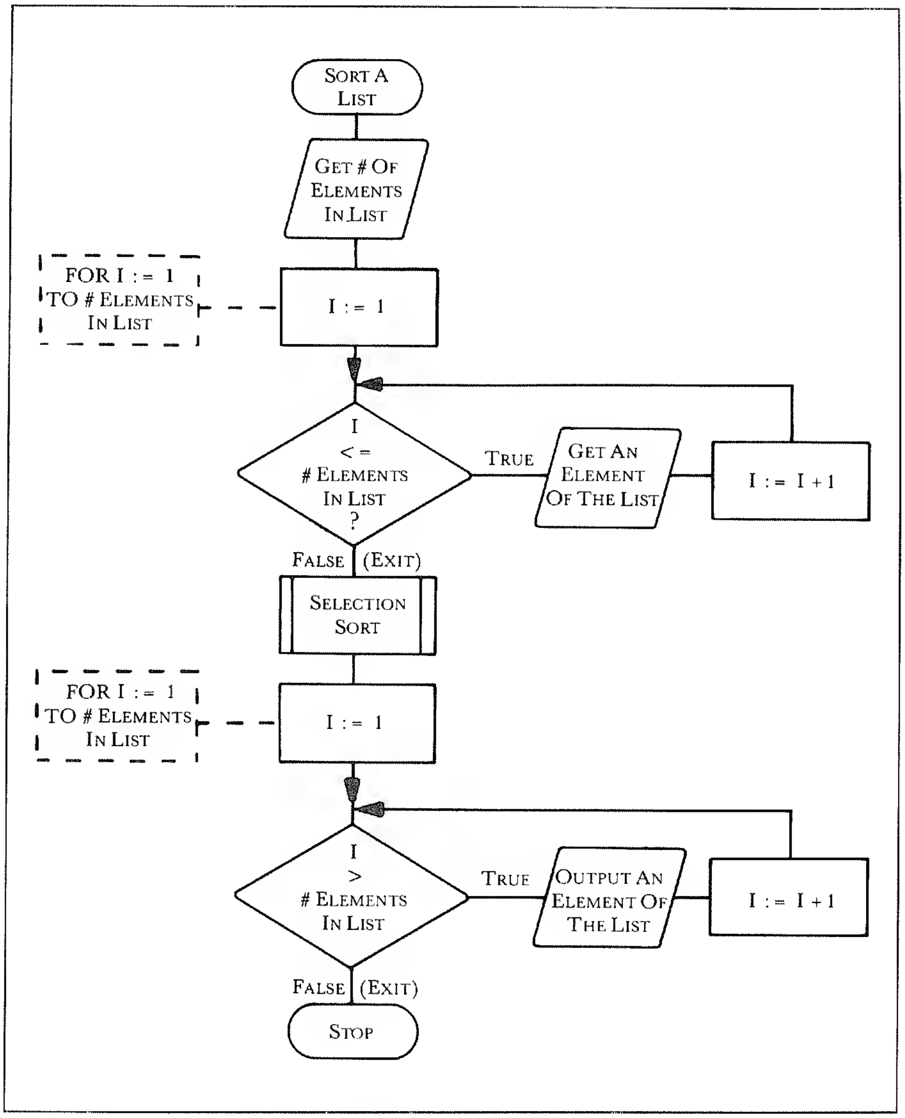

SORT A LIST
GET # OF ELEMENTS IN LIST
FOR I := 1 TO # ELEMENTS IN LIST
I := 1
I <= # ELEMENTS IN LIST ?
TRUE
GET AN ELEMENT OF THE LIST
I := I + 1
FALSE
(EXIT)
SELECTION SORT
FOR I := 1 TO # ELEMENTS IN LIST
I := 1
I > # ELEMENTS IN LIST
TRUE
OUTPUT AN ELEMENT OF THE LIST
I := I + 1
FALSE
(EXIT)
STOP

FIGURE 2.4-12: A flowchart for the *Sort-a-List* algorithm.

is changed, for instance during maintenance to add a function or correct a bug. Many programmers avoid this little chore. However, once in the habit of using pseudo-code and flowcharts as design tools, you will automatically begin with making modifications at this level before a single line of program code is changed.

3 Structured Programming

3·1 INTRODUCTION

Structured programming is concerned with improving the programming process through better program organization and better programming notation. The main concern in the early days of structured programming was the spiraling costs of program maintenance. As a result, the improvements most sought after were increases in program correctness and maintainability, rather than any direct improvement in programmer productivity or program efficiency. This led to a greater concern for the clarity of programs, since greater clarity would make it easier for the maintenance programmer to do the job.

The term structured programming got its start as a description of a style of programming that avoided GOTO statements like the plague. Indeed, the early pioneers in this area called the technique "GOTO-less programming." It was felt, and attempts were made to prove, that the vast majority of bugs in any program could be directly related to the use of GOTO statements.

However, since programming had *always* been done using GOTOs, structured programming was initially greeted as heresy. To make matters even worse, the proponents of structured programming began to talk about "structured programming languages" to replace the dinosaurs—FORTRAN, COBOL, and assembly language. There was not initially a lot of enthusiasm, either from companies or programmers, for the new methodology, or for the anticipated cost of converting to the new ideas.

Initially, the new methodology revolved around ways to facilitate correct and clear descriptions of data and control structures within a "program." In particular, the control structures of IF-THEN-ELSE, WHILE, and REPEAT-UNTIL were introduced as being more natural expressions of what actions a programmer needed performed than was the old construct of IF-GOTO. Programmers were taught to express the design of their programs in this new "language," which evolved into what we now call pseudo-code. This was used for the design of algorithms only, since no popular programming language yet existed that contained these new control structures. It wasn't until many years later that languages such as Pascal emerged from the research labs, and older languages such as FORTRAN were rebuilt to be "structured."

This meant that the wonderful, new, structured constructs had to be translated into obsolete languages that only had the IF-GOTO construct. As it turned out, this was not particularly difficult initially. The structured control constructs could all be simulated using the more primitive statement. However, the ultimate goal was to banish the GOTO from the programmer's world forever. Until the new structured languages were able to become popular, this was an idle hope. Today, most programmers can select from a wide variety of structured languages.

The most interesting thing about this bit of history, however, is that the concept of structured programming actually started out as a *design* methodology. Today, structured programming *can* achieve the long-sought goal of GOTO-less programming simply by using the structured constructs available in most modern programming languages. However, the actual *programming* part of a program's development becomes primarily a mechanical translation of the algorithm's pseudo-code.

In a sense, structured programming is the culmination of the design techniques previously discussed. It means adhering to the concepts of modularity and structured style begun during the development of the algorithms. While it might be difficult to take a very carefully structured algorithm and implement an unstructured program from it, it is certainly possible.

This chapter discusses the stylistic concerns of how to translate the structured design into appropriate code.

Disadvantages

There are several disadvantages to what might be called "strict" structuring, i.e., following every rule of the methodology blindly and precisely. This isn't to say that strict structuring should always be avoided. As you will see, there are many more advantages to the techniques than disadvantages. However, most of the rules given are really rules of thumb, guidelines to be followed as long as it makes sense to do so.

The main disadvantage to the structured techniques is that they require more initial time to design and implement the software. This was probably the biggest obstacle to be overcome by proponents of the methods trying to convince companies that there was going to be a big payoff further down the line. Perhaps even more difficult to convince were the programmers themselves. Their entire thinking had to be reworked to accept this new life cycle for programming. It is also difficult to accept the long-term benefits of such a technique without any type of feedback. Previously, feedback revolved around watching the code grow and execute, even if such execution was full of bugs.

The second most frustrating disadvantage is that structured techniques often produce code that is less efficient than could otherwise be achieved. This can result from, for instance, subroutines, created for the sake of modularity, that are never called more than once. In most applications, the increase in speed that might be achieved by not following structured techniques is negligible, and won't be noticed by users.

There are obviously applications where speed is extremely important, either because the application is time sensitive (e.g., an autopilot computer on an airplane) or because the amount of code that must be executed or the amount of data that must be manipulated makes the function noticeably sluggish. For instance, a routine to sort data might work quickly (on the order of seconds) for one hundred names and addresses, but take days to execute when the number of items to be sorted approaches one hundred thousand. In these cases, where the execution speed becomes important, it is usually better to attempt to **optimize** (make faster) a structured program than to implement an unstructured program in an effort to create one that is fast to begin with.

A third disadvantage to the structured techniques is that they sometimes may result in repeating code in a program. This may appear to be a redundancy, but generally the repeated code is unrelated to its duplicates.

Finally, structured programs have a tendency to have more code and require larger data areas than unstructured ones. Again, this trade-off is usually not very limiting. However, when memory space is limited, other types of optimizing may need to be done.

Advantages

The disadvantages discussed above are usually not overwhelming and can most often be counteracted by the advantages of using structured techniques. To begin with, the overall code of a structured program is generally more efficient than unstructured code because the overhead involved in branching can be minimized. This is achieved by eliminating GOTOs as much as possible. GOTOs are overhead because they perform no real work but require significant amounts of computer time to perform.

Next, while the structured techniques require more time in the development phase of a project, the use of such techniques invariably decreases the amount of time spent in the testing and debugging phase. Isolating and correcting errors is greatly simplified.

Finally, an even larger payoff is achieved by reducing the maintenance effort significantly. Adding new features is easier for a structured program, and the readability of the code makes it more likely that the maintenance programmer will understand it. Any start-up costs of converting to structured programming techniques are ultimately overridden by these savings.

3·2 IMPLEMENTATION GUIDELINES

Many of the guidelines that will be presented are not strictly a part of what is currently called structured programming. They fall more into a category of "good programming practices," and often are simply rules of thumb applied whenever it seems practical to do so. Some of the guidelines should strike you as common sense. Others are dictated by peculiarities of a programming language like BASIC.

If you have followed the formulae given so far in this book, you should be ready to begin coding your program. This can be done in a top-down fashion, bottom-up, or with a combination of the two methods. As discussed in detail in Chapter 5, Program Testing and Debugging, each approach has its merits. You are probably wisest simply to continue to use whatever approach you used during the design phase of the project. While this is not in any way a requirement, it is perhaps the most natural approach.

In this chapter we are (finally!) going to be concerned with details of the implementation. In particular, this section discusses rules for coding that have evolved from the structured programming principles. In addition, it offers tips for making the coding process simpler and less error prone. Although, after following the methods outlined previously in this book, the actual coding becomes a somewhat mechanical task, errors do crop up in the translation, just as they invariably do when translating natural languages.

Modules

We will first look at modules. While modules have already been defined during the design phase, further care must be taken when converting these designs into actual code. In addition, the term *module* has a slightly different connotation when dealing with the actual coded program. It usually refers to a subroutine, specifically one that has been compiled separately from the rest of the program. This would be the case, for instance, when dealing with a library of routines, as discussed later in the Libraries section.

However, the term can also be applied to any collection of statements that follow a particular philosophy. This philosophy is described in detail below, and includes principles concerning how code is formed into coordinated collections. These principles deal primarily with the functions that the module performs and the proper interfacing of modules, and are more detailed than what was described in section 1.3, Modular Design, because we were not concerned with such details during the preliminary design. At that time, we were only concerned with defining very high-level functions.

The principles described are applied, in some degree, to every collection of code in a program. Although there are some feelings among professionals that the size of a module (i.e., the number of lines of code) should be kept to a particular maximum, the other principles that are discussed are usually applied not only to subroutines, but also to individual control structures as well as to the program as a whole. This is vital for ensuring consistency in the code.

In fact, the use of pseudo-code for developing algorithms is a direct derivative of many of the philosophies described below. You should be able to easily recognize how pseudo-code plays this role for each of the principles discussed below.

Uni-Functionality

The first principle is that each module should have a single, well-defined purpose. This means that the module should perform only a single function. This is easily understood for the lowest level modules that were defined during the preliminary design. It is perhaps more difficult to understand for the higher-level modules, since they are responsible for all of the functions defined below them. However, that is precisely the one function that these higher-level modules have—to orchestrate the functions below them.

This is not as capricious as it might first appear. Recall that during the preliminary design phase we had defined a module as a portion of the program, such as a block of code or a subroutine, that performs a specific function, and that we had used HIPO charts to break the program down into its functional components. The blocks defined using this technique should naturally adhere to this uni-functionality principle.

One in–One out

This principle has to do with how the code of a module is interfaced with other blocks of code. Since it deals with details of the code, it was not discussed during the preliminary design.

The one in–one out principle requires that there be only one way into a module, and only one way out. This makes it much simpler to trace the code manually, as will be required during the testing and debugging phase (see Chapter 5, Program Testing and Debugging). In addition, it preserves our previous principle of uni-functionality by making it impossible for the same section of code to be used in more than one way.

The most notorious example of multiple paths in and out of a block of code are BASIC subroutines. Since there is no defined head statement identifying the first statement in a BASIC subroutine, the program can call the routine using any of the routine's line numbers. For instance, consider the subroutine shown in Figure 3.2-1. Can you determine the function of this routine? It appears that it will add ten numbers entered by the user. But what if the call to this subroutine were GOSUB 620, instead of the assumed GOSUB 610?

```
610 TOTAL = 0
620 FOR I = 1 TO 10
630   INPUT "ENTER A NUMBER: "; X
640   TOTAL = TOTAL + X
650 NEXT I
699 RETURN
```

FIGURE 3.2-1: A subroutine that appears to sum up ten new numbers.

This causes the subroutine to perform an entirely different function, namely adding ten numbers to the previous TOTAL.

The same is true for the RETURN statement with regard to the "one out" principle. In BASIC, there can be more than one RETURN statement in a subroutine. In addition, the RETURN does not even identify the end of the subroutine.

Such confusing code can make debugging a program difficult at best. As a result, the quality of a program can only be regarded as significantly lowered by such lapses in structure. In languages such as BASIC where the structure is essentially nonexistent in the language itself, the programmer will have to impose a structure. Suggestions for doing this with BASIC are discussed later in the Data Structures section.

Module Size

One of the quickest indicators of whether you have followed the modularity philosophy sufficiently is the size (i.e., the amount of code) of the modules you have defined. It should be possible to estimate the size of each module based upon its algorithm. Occasionally, however, it may not become apparent that the size of a module is out of hand until the module is coded.

A general rule of thumb is that a module should be limited to approximately one hundred lines of executable code. Modules that are larger than this can usually be broken down further into submodules. This is a reasonable size for a module because it will generally fit on two pages of a source code listing, meaning you won't need to flip pages back and forth while reading through the code. In addition, the smaller the module, the more likely it is that you will be able to understand all of the code. The module becomes more readable and, therefore, more understandable. Anything that makes the code more readable is definitely an advantage.

Data Structures

It is wise to isolate all things dependent on a particular data structure (an array or file record) to as few modules as possible. This makes it far easier to change the data structure's format should the specifications of the program change.

For instance, consider a record format for a particular file. If a new field must be added to the record (e.g., the extra four digits of the new nine-digit zip code), having the fields of the record used in a limited number of modules makes the addition much less painful.

This leads to another sound practice for dealing with files. When possible, create separate modules for doing input and output for a particular file, and use these modules only to perform I/O on that file. This makes certain that the record format is handled correctly every time I/O is performed on that file. This is especially important in a language such as BASIC where there is no required record format for a file and, therefore, no mechanism to detect any violation of the file's format.

Creating I/O modules works well with inputs and outputs from/to the user when a particular type of input or output must occur in more than one place within the program. Figure 3.2-2 gives an example of an output subroutine that generates a line of asterisks on the screen. A call to this subroutine can be made any time this action is desired. Care must be taken, however, not to create a module that causes more trouble than it avoids, as is discussed in the following section.

Duplicate Functions

In creating modules, one object is to eliminate duplicate functions from the program, replacing the code of the duplicates with calls to

FIGURE 3.2-2: A BASIC subroutine that generates a line of asterisks.

```
600 REM Output a blank line, a row of asterisks,
605 REM and another blank line
610 PRINT
620 FOR I6% = 1 TO 80
630   PRINT "*";
640 NEXT I6%
650 PRINT
660 PRINT
699 RETURN
```

a module that performs the common function. However, every case of duplicate code should not necessarily be eliminated. Separating a "random" collection of code into a module simply because it would be repeated in another section of the program is not a good idea if that code does not perform a specific, definable function that is common to the two areas of code. The problem is that separating the code out into a subroutine creates a dependency on this code at each place where a call to the subroutine is made. If there were really no relationship between the two portions of the program other than the repetition of several lines of code, trouble occurs when one of the sections needs to be changed during maintenance. If you change the subroutine, the other section of code—the one not requiring the change—is also affected.

Consider the code in Figure 3.2-3. This program segment performs two functions. First, it reads in a list of numbers and calculates their sum. Then it reads in another list of numbers and calculates the product. There appears to be an obvious duplication, since the control structures defined are identical in both parts of the program. There are only some slight differences in the use of variables. Wouldn't this all be easier to code, therefore, by writing a single subroutine that is called twice, once for calculating the sum, and another time for calculating the product?

Figure 3.2-4 shows how a subroutine can be devised which gives

FIGURE 3.2-3: A program segment that appears to have a duplicated function that could be separated out into a separate module.

```
500 INPUT "How many items to be added? ", NITEMS
510 TOTAL = 0
520 FOR I = 1 TO NITEMS
530   INPUT "Enter an item: ", ITEM
540   IF ITEM <= 0 THEN
          PRINT "*** Item must be greater than zero":
          GOTO 530
550   TOTAL = TOTAL + ITEM
560 NEXT I
570 PRINT
580 INPUT "How many items to be multiplied? ", NITEMS
590 PRODUCT = 1
600 FOR I = 1 TO NITEMS
610   INPUT "Enter an item: ", ITEM
620   PRODUCT = PRODUCT * ITEM
630 NEXT I
640 PRINT TOTAL, PRODUCT
```

```
520 WHAT$ = "added"
530 GOSUB 910
540 TOTAL = RESULT
545 PRINT
550 WHAT$ = "multiplied"
560 GOSUB 910
570 PRODUCT = RESULT
580 PRINT TOTAL, PRODUCT
      .
      .
      .
910 PRINT "How many items to be ";WHAT$;" ";
920 INPUT NUM
930 IF WHAT$ = "added" THEN
          RESULT = 0
    ELSE RESULT = 1
940 FOR I = 1 TO NUM
950   INPUT "Enter an item: ", ITEM
960   IF WHAT$="added" THEN
            RESULT=RESULT + ITEM
      ELSE RESULT=RESULT * ITEM
970 NEXT I
979 RETURN
```

FIGURE 3.2-4: A subroutine can be used to eliminate the apparently redundant code in Figure 3.2-3. This code provides exactly the same functionality as the previous code.

the exact same functionality of the code in the previous figure. The question we are faced with is whether or not this is a good idea. It seems to save code by eliminating a duplicate function, although in a different way than that shown in Figure 3.2-2.

The question of whether this constitutes a duplicate function or just duplicate code cannot be easily determined. There are two rules of thumb, however, that can usually be applied for making this decision. The first is that the newly created subroutine should adhere to the guidelines of a module. In this case, the applicable consideration is: Does this routine have a single function that is easily definable? The answer is no, since the routine either calculates a sum, or calculates a product, depending on how it is called.

The second heuristic that can be applied is that a change in the requirements of one of the functions should not adversely affect the other function. For example, what would happen if we wanted to restrict the product to numbers greater than zero, but wanted to be able to sum any numbers? This is clearly a problem in this example, and is evident from the special code required in lines 930 and 960 of the subroutine to differentiate between using this code for calculating a sum or a product.

These difficulties reveal that this subroutine has eliminated duplicate code but has *not* created a module to perform a common function.

Clever programming tricks such as this may look neat, but they generally cause more trouble than they are worth, and ultimately wind up costing the programmer more time than they supposedly would save.

Libraries

Once you have taken the trouble to create sophisticated routines for one program, it seems a shame to have to create entirely new routines for other programs that perform the same types of functions. This "reinventing the wheel" syndrome can be overcome by creating a library of routines that would be useful in more than one situation.

Large-scale computer companies have been doing this for many years. For instance, IBM has a package called the Scientific Subroutine Package (SSP) that contains dozens of modules that can be called from many different languages to perform standard scientific calculations. A programmer could create a program that would call this SSP in a special way to perform one of its functions on data that the program would supply. This saves the programmer much work in writing and rewriting these functions every time he needs them.

Creating such a generic package of routines is extremely complex. However, it is surprising how many times you want to include a similar function in more than one program. In addition, it isn't nearly as difficult to create routines for use by a single language as it is to make the routines accessible to many languages.

For BASIC, it is possible to set up a routine in its own file. When you want to use this routine, you would copy the file into the program you want to use it in. This is generally done with a command such as MERGE, which copies one BASIC program file into another, and allows you to use the code that was copied as if it were originally a part of your program.

This merging can be done when you are creating the source code, making the merged code a permanent part of your program, or it can be done each time your program is executed. Figure 3.2-5 shows how this can be done using Microsoft BASIC on the DEC Rainbow. In (b) is the routine that is to be copied into the program, shown in (a). The subroutine (b) must have been saved with the "A" option,

```
10 CHAIN MERGE "DUMMY.BAS",20
20 PRINT "MERGE DONE"
30 GOSUB 100
40 PRINT A,B
50 STOP
999 END
```

a

```
100 A=100
110 B=5
120 PRINT "END OF DUMMY"
130 RETURN
```

b

FIGURE 3.2-5a and b: An example of the CHAIN facility in Microsoft BASIC. This is one method used to create a module library.

which stores the BASIC program file in ASCII form. Line 10 in (a) performs the copy of the routine which was stored with the name "DUMMY. BAS". The 20 at the end of line 10 tells what line execution should continue on after the merge is completed.

Figure 3.2-6 shows the results of the merge, and the output generated by this example. After the merge is completed, BASIC

FIGURE 3.2-6: The results of the execution of the BASIC code in Figure 3.2-5a. The listing shows the result of the merge, which is done when the code in Figure 3.2-5a was RUN, and the output generated hy the execution.

```
10 CHAIN MERGE "DUMMY.BAS",20
20 PRINT "MERGE DONE"
30 GOSUB 100
40 PRINT A,B
50 STOP
100 A=100
110 B=5
120 PRINT "END OF DUMMY"
130 RETURN
999 END


MERGE DONE
END OF DUMMY
 100           5
Break in 50
Ok
```

```
program chain(input,output);

var a, b: integer;

{$I #5:DUMMY}

begin
  dummy;
  writeln(a,'   ',b)
end.
```

a

```
procedure dummy;
begin
  a := 100;
  b := 5;
  writeln('end of dummy')
end;
```

b

```
Pascal Compiler IV.1 c5s-4      8/23/84


    1   2    1:d    1  program chain(input,output);
    2   2    1:d    1
    3   2    1:d    1  var a, b: integer;
    4   2    1:d    3
    5   2    1:d    3  procedure dummy;
    6   2    2:0    0  begin
    7   2    2:1    0    a := 100;
    8   2    2:1    4    b := 5;
    9   2    2:1    7    writeln('end of dummy')
   10   2    1:0    0  end;
   11   2    1:0    0
   12   2    1:0    0
   13   2    1:0    0
   14   2    1:0    0  begin
   15   2    1:1    0    dummy;
   16   2    1:1    2    writeln(a,'   ',b)
   17   2     :0    0  end.

End of Compilation.
```

c

FIGURE 3.2-7a, b, and c: An example of the use of the UCSD p-system *include* facility. The text enclosed in the brackets ({}) in (a) is a compiler instruction (called a pseudo-comment) to find the file called #5:DUMMY and copy the text in that file into this program. The text of this file is shown in (b). The result can only be seen by looking at the compiler listing, shown in (c). We can see here that the file was copied into the original source code, without any trace that it was ever a separate file.

treats the new code as if it had been in the program to begin with.

In Pascal, a similar facility exists to copy code into a program, but it must be done at compile time, not execution time. The result of using this **include** facility is similar to that shown in the above BASIC example, except that copying occurs only once, during compilation, not every time the program is executed. Figure 3.2-7 shows an example of the include mechanism using UCSD Pascal.

Another mechanism for obtaining similar results is called **linking.** This requires a special program called a **linker,** which is included with most compilers. The function of the linker is to create a single executable program for multiple, separately compiled modules. Such a facility is different from the include mechanism in that the code that is linked has already been compiled, and so is in machine language.

This makes it possible to use sophisticated functions without having to worry about the source code for these functions. It also means that, when the module that is being linked with your program is changed some time in the future, you only have to relink the new version with your already compiled program, not recompile the entire thing.

Obviously, great care must be taken when using such a facility, since rigorous syntax considerations apply. However, much time can be saved by creating files with generic routines to be copied into or linked with new programs. The types of functions that you would want to include in a library will depend greatly on the types of applications you personally develop. Examples are scientific routines, sorting routines, and specialized input or output routines. The routines do not necessarily need to be completely ready to execute, as in the examples above. Instead, they could be copied into the source code of your program and there customized for the particular application required in that program.

Variables

Second only to the unrestricted use of GOTOs, the misuse of variables and their names probably causes more bugs than any other problem. It undoubtedly stems from variables being somewhat unobtrusive in a program filled with control structures, data structures, files, etc. However, trying to locate a misspelled variable name can be more frustrating and time-consuming than anything that might go wrong with these other constructs. Following are some guidelines for dealing with variables.

Descriptive Names

One of the easiest traps to fall into when coding a program is to use very simple variable names. Perhaps we have been somewhat brainwashed into using names such as X and Y by our experience in mathematics, where we must often deal with equations of the form $ax^2 + bx + c = 0$. Even in dealing with the infamous "word problem" in algebra, we were taught to use single letter names for variables. At best, we were allowed to use something like X_3.

The biggest problem most students have with algebra, however, seems to stem from not being able to make the mental connection between a variable named X and its meaning. This comes as little surprise, since the same variable named X is often used in every problem to be solved. As a result, many people have tremendous difficulty using algebraic techniques, even though using them would make many tasks much simpler.

This problem is magnified when dealing with programming, where dozens of variables may be used in a single program. How could anyone consistently remember that in one program X stands for a temperature, while in another program X is a length?

The way out of this mess is obvious. The name of the variable should in some way indicate its purpose. This means that if a variable is needed to hold the value of a temperature, a reasonable name for that variable would be TEMPERATURE. If there needs to be more than one variable for temperature, it makes sense to call such variables TEMPERATURE1 and TEMPERATURE2. In this way, it is (supposedly) obvious when reading the code exactly what the purpose of each variable is.

In the recent past, it has been impossible to name variables with descriptive names because the languages limited the length of all names quite severely. In the earliest versions of BASIC, for instance, names could be only two characters long, the first being a letter, and the second being a number, as in T1 and T2.

Fortunately, most modern languages have overcome this deficiency, and allow names to be as much as forty or more characters in length. Unfortunately, not *all* languages currently being used allow longer names. There are many versions of BASIC and FORTRAN on large computers that limit names to between two and six characters. In Applesoft BASIC on the Apple IIe, only the first two characters of a name count for uniqueness of the variable name, although names can be longer than two characters.

Even when variable names can be long enough to be descriptive, they can also present a problem when they get to be too long. Consider

NEW.BALANCE.ACCOUNTS.PAYABLE

and

NEW.BALANCE.ACCOUNTS.RECEIVABLE

as two names in the same program. These are nicely descriptive names. But if they appear more than a couple of times in the program, you'll quickly get tired of writing them out.

The usual remedy to this is to use abbreviations. Thus, you might end up with

NEW.BAL.AP

and

NEW.BAL.AR

instead of the names given above. These are moderately descriptive, even for someone who is not intimately familiar with the program. Once a newcomer is told what these names stand for (e.g., in the comments), he or she should have little trouble remembering their purpose.

There is a danger in using abbreviations, however. In the above example, the danger is that one of the names will be misread by a programmer skimming the program's listing. The two names look too much alike. This could lead to a lot of confusion and lost time. Even worse, the listing might be a little obscure because of a bad printing. For instance, if the listing were printed using a dot matrix printer with low resolution, the final R in NEW.BAL.AR might come out looking like a P.

The second problem is that an abbreviation might be misunderstood, even if it is explained clearly in comments elsewhere in the program. A programmer might not bother to look up the meaning of an abbreviation for a variable whose name seems familiar. For instance, if we were to use the name TEMP in a program, a programmer might assume that this is an abbreviation for "temperature," when in fact it stands for "temporary." This confusion could lead to many hours of extra work.

There is not an easy solution to this dilemma. Names must be long enough to be clearly understood. In addition, you should avoid using names that are too similar to one another. Finally, all names should be fully described in the documentation of the program (this is discussed in more detail in section 3.3, Program Style and Chapter 6, Documentation).

Multipurpose Variables

One of the most devious things that a programmer can do with a simple variable is to use it for more than one purpose within the same program. For instance, imagine that the programmer uses the variable TEMP first as a temperature. Later in the program, when he no longer needs a variable to hold a value for a temperature, he reuses the name TEMP to hold a temporary value.

This sets up any maintenance programmer for a lot of misery. Imagine reading along in this program, knowing that the variable named TEMP holds a temperature value, and suddenly running across a line of code where TEMP is used to hold a temporary value, such as in

```
800 LET TEMP = X
810 LET X = Y
820 LET Y = TEMP
```

You would probably wonder what relationship a *temperature* has to variables X and Y! The correct answer is "none." This type of switch in the meaning of a variable leads to confusion later on, whether or not the original programmer is doing the maintenance.

The rule here is that a variable name should only be used for one purpose throughout the entire program. For instance, once the name TEMP has been used to mean a temperature, another name should be selected for the temporary variable. While this might mean using additional space in the program to hold variables that are no longer being used, the amount of actual waste is negligible.

This is generally not a problem if the previous rule about descriptive names is followed carefully. However, it is still tempting to reuse some common variable names. For instance, I generally give loop control variables names such as I or J. These names can be reused in later portions of the program, as long as they are used strictly for loop control. The example in Figure 3.2-3 shows this with the variable I. As long as the number of exceptions to this multi-use prohibition is kept reasonably small, there are usually few problems.

Real (Floating Point) Values

In most languages, there is a complex distinction between integer variables and variables that may contain fractional (decimal) values. These latter values are called **floating point** values in BASIC and **real** values in Pascal. The way variables of this type are stored differs from language to language (and sometimes version to version). In general, the internal representation of such variables follows a pattern that includes a **sign bit,** a **mantissa,** and an **exponent.** A fairly standard length for a floating point number in memory is 32 bits, or 4 bytes.

The sign bit simply indicates whether the value currently stored in this location is positive or negative. If the bit is 0, then the number is positive; a 1 indicates negative.

The mantissa is a special format of the actual number being stored. A typical size for the mantissa might be 24 bits, giving the maximum value of the mantissa as 33554431 decimal. The special format is called a **normalized** form, meaning that the decimal point is understood to be to the left of the mantissa. This makes the number being stored 0.33554431, rather than as 33,554,431.

It is the third part, the exponent, that gives the floating point value its large range. A typical exponent will be 7 bits long, 1 bit for a sign (positive or negative exponent) and 6 bits for the actual value of the exponent. This means that the maximum positive exponent would be 63, while the maximum negative exponent would be 64.

The exponent is used to determine exactly where the decimal point should be placed within the actual number. The actual value of the number being stored is calculated as

$0.\text{mantissa} * 10^{\text{exponent}}$

For example, if the mantissa being stored were 12345, and the exponent were +3, then the value being stored is

$0.12345 * 10^{+3}$

which is 123.45 in decimal. If the exponent had been −3 instead of +3, then the actual value would be 0.00012345, since 10 raised to the power of a negative exponent moves the decimal point to the left. Note that the exponent moves to the right or the left exactly the number of places indicated by the exponent.

It is not important that you know exactly how this works. What is important is that you understand two details of the mechanism. First, the fixed lengths of the mantissa and the exponent determine the magnitude of the value that can be stored. With the above format, the largest value that can be stored is $0.33554431 * 10^{63}$. The smallest number is $0.33554431 * 10^{-64}$. This means that, while the largest and smallest values might be dozens of digits long, *the maximum accuracy of any number is at most eight digits!* As a result, any value that requires greater accuracy is truncated when it is stored in memory, changing the value.

For an example of how this causes trouble, refer to Figure 3.2-8. The statements in lines 10 through 50 assign **double precision** (a floating point number stored in 8 bytes instead of the usual 4) numbers, indicated by #, into single precision (4-byte floating point) variables. When the values of these single precision variables are later printed out, you can see how they have been changed by truncation. The best precision that can be achieved in a single precision variable with this version of BASIC is seven digits. All other digits are effectively lost.

Second, another type of precision loss is also occurring in this example. Note the difference in the output values for X2 and X3. Then look at X3 and X4. Even though the difference between X3 and X4 is really greater than that between X2 and X3, the output

FIGURE 3.2-8: An example of loss of precision when assigning a number with many decimal places to a single precision variable in BASIC. The bottom of the figure shows the output generated by running the code. Compare the values that were output with the values assigned to the variables in lines 10 through 50.

```
10 X1 = 987654301#
20 X2 = 987654303#
30 X3 = 987654304#
40 X4 = 987654399#
50 X5 = 987654499#
60 PRINT "X1 = "; X1, "X2 = "; X2
70 PRINT "X3 = "; X3, "X4 = "; X4
80 PRINT "X5 = "; X5
90 END




X1 =  9.876543E+08              X2 = 9.876543E+08
X3 =  9.876544E+08              X4 = 9.876544E+08
X5 =  9.876546E+08
```

values show only a difference between X2 and X3, but not between X3 and X4. In addition, note the large difference in the output value of X5 in comparison to the value assigned to it in line 50.

This inaccuracy is due to another type of error that occurs when dealing with floating point values. The problem is that values are stored in binary, not in decimal. We will not go into the details of the mechanism for the conversion between binary and decimal, or vice versa. It is sufficient to know that this conversion is not exact when converting decimal fractions to binary. Some decimal values cannot be precisely represented in binary. As a result, these values suffer from approximation error when they are reconverted to decimal for, say, outputting.

Consider the program and its output in Figure 3.2-9. In this

FIGURE 3.2-9: An example of the inaccuracy of floating point numbers.

```
10 COUNT = 0
20 FOR I = 0 TO 1 STEP .01
30   PRINT I,
40   COUNT = COUNT + 1
50   IF COUNT = 4 THEN PRINT: COUNT=0
60 NEXT I



0             .01           .02           .03
.04           .05           .06           .07
7.999999E-02  8.999999E-02  9.999999E-02  .11
.12           .13           .14           .15
.16           .17           .18           .19
.2            .21           .22           .23
.2400001      .25           .26           .27
.28           .29           .3            .31
.32           .33           .34           .3499999
.3599999      .3699999      .3799999      .3899999
.3999999      .4099999      .4199999      .4299999
.4399999      .4499999      .4599998      .4699998
.4799998      .4899998      .4999998      .5099998
.5199998      .5299998      .5399998      .5499998
.5599998      .5699998      .5799998      .5899998
.5999998      .6099997      .6199997      .6299997
.6399997      .6499997      .6599997      .6699996
.6799996      .6899996      .6999996      .7099996
.7199996      .7299996      .7399996      .7499996
.7599996      .7699996      .7799996      .7899996
.7999995      .8099995      .8199995      .8299995
.8399995      .8499995      .8599995      .8699994
.8799994      .8899994      .8999994      .9099994
.9199994      .9299994      .9399994      .9499994
.9599994      .9699994      .9799994      .9899994
.9999994
```

elementary BASIC program, all values between 0 and 1 are printed out by increments of 0.01. Notice what happens to certain values. The value 0.08 cannot be represented accurately in binary, so the resulting value of 7.999999E-02 is output. In looking at all these values, it is somewhat shocking to see that most cannot be represented precisely. In some of the values, the loss of precision is quite dramatic.

Figure 3.2-10 shows a similar result for a program written in UCSD Pascal. While the values here are more precise than in the BASIC example, approximation error still occurs and can, therefore, cause much havoc.

These types of errors can also be introduced when combining integers and real values in the same expression. An example is the expression

```
LET X = 1/3.0
```

in BASIC, or

```
X := 1/3.0
```

in Pascal. The language must convert any integers into floating point representations before the actual calculations can be performed. In this case, the 1 is an integer and must be converted to a floating point representation before the division can be performed. This conversion routine can introduce additional errors, resulting in further loss of precision.

Unless specific steps are taken to increase the precision of a floating point variable, its accuracy will be limited in this way. In BASIC, all numeric variables are stored as single precision floating point values unless they are explicitly identified as some other type. The other three standard types of variables are strings, integers, and double precision floating point. To indicate that a variable is to have a string value, the variable name is appended with a dollar sign ($), e.g., ADDRESS$. Integer variables are identified by appending a percent sign (%), e.g., I%. Double precision variables are identified with the pound sign (#), e.g., BUDGET#.

This means that unless you have identified variables using these special symbols, all numeric values are stored as single precision numbers, and are subject to the difficulties outlined above. This does not usually cause a problem or a significant loss of accuracy when making simple calculations. Most errors introduced in this way will

```
program precise(input,output);
var i: real;
    j: integer;
begin
  i := 0.0;
  j := 0;
  while (i < 1.0) do
  begin
    write(i,'     ');
    i := i + 0.01;
    j := j + 1;
    if j = 3 then
    begin
      writeln;
      j := 0
    end
  end
end.
```

```
0.000000000000000      1.000000000000000E-2      2.000000000000000E-2
3.000000000000000E-2      4.000000000000000E-2      5.000000000000000E-2
6.000000000000001E-2      7.000000000000001E-2      8.000000000000000E-2
9.000000000000000E-2      1.000000000000000E-1      1.100000000000000E-1
1.200000000000000E-1      1.300000000000000E-1      1.400000000000000E-1
1.500000000000000E-1      1.600000000000000E-1      1.700000000000000E-1
1.800000000000000E-1      1.900000000000000E-1      2.000000000000000E-1
2.100000000000000E-1      2.200000000000001E-1      2.300000000000001E-1
2.400000000000001E-1      2.500000000000000E-1      2.600000000000000E-1
2.700000000000001E-1      2.800000000000001E-1      2.900000000000001E-1
3.000000000000001E-1      3.100000000000001E-1      3.200000000000001E-1
3.300000000000001E-1      3.400000000000001E-1      3.500000000000001E-1
3.600000000000001E-1      3.700000000000001E-1      3.800000000000002E-1
3.900000000000002E-1      4.000000000000002E-1      4.100000000000003E-1
4.200000000000002E-1      4.300000000000003E-1      4.400000000000003E-1
4.500000000000003E-1      4.600000000000003E-1      4.700000000000003E-1
4.800000000000003E-1      4.900000000000003E-1      5.000000000000003E-1
5.100000000000003E-1      5.200000000000003E-1      5.300000000000003E-1
5.400000000000003E-1      5.500000000000003E-1      5.600000000000004E-1
5.700000000000003E-1      5.800000000000004E-1      5.900000000000003E-1
6.000000000000004E-1      6.100000000000004E-1      6.200000000000004E-1
6.300000000000004E-1      6.400000000000004E-1      6.500000000000004E-1
6.600000000000004E-1      6.700000000000004E-1      6.800000000000005E-1
6.900000000000004E-1      7.000000000000005E-1      7.100000000000004E-1
7.200000000000005E-1      7.300000000000005E-1      7.400000000000005E-1
7.500000000000005E-1      7.600000000000005E-1      7.700000000000005E-1
7.800000000000006E-1      7.900000000000005E-1      8.000000000000005E-1
8.100000000000004E-1      8.200000000000004E-1      8.300000000000006E-1
8.400000000000005E-1      8.500000000000005E-1      8.600000000000004E-1
8.700000000000006E-1      8.800000000000006E-1      8.900000000000005E-1
9.000000000000005E-1      9.100000000000006E-1      9.200000000000006E-1
9.300000000000006E-1      9.400000000000005E-1      9.500000000000007E-1
9.600000000000006E-1      9.700000000000006E-1      9.800000000000006E-1
9.900000000000007E-1
```

FIGURE 3.2-10: A Pascal program that outputs all the values between 0.0 and 1.0, using an increment of 0.01. Note that most of the values are not precise. This is generally caused either by errors occurring when converting from decimal to binary, or by the inability of a particular decimal value to be exactly represented as a binary value within the precision of the Pascal type *real*.

be eliminated by simple formatting of the output using, for instance, a PRINT USING statement.

Where these representations really cause trouble is in making comparisons. Whenever a floating point variable is involved in a comparison, its imprecise value can change the outcome. Consider the programs shown in Figures 3.2-11 and 3.2-12. Algebraically, the value of the variable J should exactly equal I, since (1/I)*I evaluates to 1, and 1*I evaluates to I. However, because of conversions and storage imprecisions, J occasionally does *not* equal I. This happens more frequently in the BASIC example than in the Pascal because of the differences in the way each language handles numbers.

Logically, these programs should have produced 100 "yes" outputs. But they didn't. How would you ever tell this by looking at the listing?

There are two ways to protect against this type of error. First, explicitly define every variable to be of a specific type. In Pascal this is required, but in BASIC it is not. Every variable in BASIC is assumed to be a single precision number unless specified otherwise by the flags $, %, and #. Therefore, use these flags religiously to make certain that integers are treated as integers. Simple counters should never be floating point values. Not only will this help protect you from errors, but it will also increase the efficiency of your code.

The second method of protection is to never directly compare floating point values. Instead, test for the difference of two values to a **tolerance level** of exactness. This tolerance level tells how close the two values need to be in order to be considered equal. How large or small this tolerance level is will be dependent upon the amount of accuracy required by the particular application, as well as the precision of the numbers being used.

Figure 3.2-13 shows how the previous BASIC program could be repaired so that it gives the correct answer every time. In this case, a tolerance level of 0.001 was used. This means that as long as the values of I and J are within one one-thousandth of one another, they are considered to be "equal."

Although the error appears to occur in about the fifth decimal place, it would be somewhat dangerous to use a tolerance level much smaller than that used. The rule of thumb here is that the tolerance level should be at least one order of magnitude (i.e., ten times) larger than the possible error. In this example, the error probably occurred in the ten-thousandths place. Therefore, we used a tolerance level of one one-thousandth, or ten times one ten-thousandth. We should

```
10 K = 0
20 FOR I=1 TO 100
30    J = ((1/I)*I)*I
35    PRINT TAB(K*23);
40    IF I = J THEN
           PRINT "yes:  ";
      ELSE PRINT "no:   ";
50    PRINT I;" "; J;
60    K = K + 1
70    IF K = 3 THEN PRINT: K = 0
80    NEXT I
90 END
```

```
yes:   1    1            yes:   2    2            yes:   3    3
yes:   4    4            yes:   5    5            yes:   6    6
yes:   7    7            yes:   8    8            yes:   9    9
yes:   10   10           yes:   11   11           yes:   12   12
yes:   13   13           yes:   14   14           yes:   15   15
yes:   16   16           yes:   17   17           yes:   18   18
yes:   19   19           yes:   20   20           yes:   21   21
yes:   22   22           yes:   23   23           yes:   24   24
yes:   25   25           yes:   26   26           yes:   27   27
yes:   28   28           yes:   29   29           yes:   30   30
yes:   31   31           yes:   32   32           yes:   33   33
yes:   34   34           yes:   35   35           yes:   36   36
yes:   37   37           yes:   38   38           yes:   39   39
yes:   40   40           no:    41   41           yes:   42   42
yes:   43   43           yes:   44   44           yes:   45   45
yes:   46   46           no:    47   47           yes:   48   48
yes:   49   49           yes:   50   50           yes:   51   51
yes:   52   52           yes:   53   53           yes:   54   54
no:    55   55           yes:   56   56           yes:   57   57
yes:   58   58           yes:   59   59           yes:   60   60
no:    61   61           yes:   62   62           yes:   63   63
yes:   64   64           yes:   65   65           yes:   66   66
yes:   67   67           yes:   68   68           yes:   69   69
yes:   70   70           yes:   71   71           yes:   72   72
yes:   73   73           yes:   74   74           yes:   75   75
yes:   76   76           yes:   77   77           yes:   78   78
yes:   79   79           yes:   80   80           yes:   81   81
no:    82   82           no:    83   83           yes:   84   84
yes:   85   85           yes:   86   86           yes:   87   87
yes:   88   88           yes:   89   89           yes:   90   90
yes:   91   91           yes:   92   92           yes:   93   93
no:    94   93.99999 yes:   95   95           yes:   96   96
no:    97   96.99999 yes:   98   98           yes:   99   99
yes:   100   100
```

FIGURE 3.2-11: Although algebraically the expression in line 30 evaluates exactly to I, note that when dealing with a program, I does not equal J in many instances.

```
program nothing(input,output);
var i, j: real;
    k: integer;
begin
  for k := 1 to 100 do
  begin
    i := k;
    j := ((1/i)*i)*i;
    if i = j then write('yes: ') else write('no:  ');
    writeln(i,'    ',j)
  end
end.



yes:  1.000000000000000     1.000000000000000
yes:  2.000000000000000     2.000000000000000
yes:  3.000000000000000     3.000000000000000
                     .
                     .
                     .

yes:  4.300000000000000E1     4.300000000000000E1
yes:  4.400000000000001E1     4.400000000000001E1
yes:  4.500000000000000E1     4.500000000000000E1
yes:  4.600000000000000E1     4.600000000000000E1
yes:  4.700000000000001E1     4.700000000000001E1
yes:  4.800000000000000E1     4.800000000000000E1
no:   4.900000000000001E1     4.900000000000000E1
yes:  5.000000000000000E1     5.000000000000000E1
yes:  5.100000000000000E1     5.100000000000000E1
yes:  5.200000000000001E1     5.200000000000001E1
                     .
                     .
                     .

yes:  9.000000000000000E1     9.000000000000000E1
yes:  9.099999999999999E1     9.099999999999999E1
yes:  9.199999999999999E1     9.199999999999999E1
yes:  9.300000000000000E1     9.300000000000000E1
yes:  9.400000000000000E1     9.400000000000000E1
yes:  9.500000000000000E1     9.500000000000000E1
yes:  9.599999999999999E1     9.599999999999999E1
yes:  9.699999999999999E1     9.699999999999999E1
no:   9.800000000000000E1     9.799999999999998E1
yes:  9.900000000000000E1     9.900000000000000E1
yes:  1.000000000000000E2     1.000000000000000E2
```

FIGURE 3.2-12: A Pascal program showing how calculations with *real* variables can cause imprecision. Algebraically, every comparison should have resulted in an output of "yes". Note that this routine performs better than its BASIC counterpart, but still fails twice.

```
10 K = 0
20 FOR I=1 TO 100
30    J = ((1/I)*I)*I
35    PRINT TAB(K*23);
40    IF ABS(I - J) < .001 THEN
             PRINT "yes:  ";
      ELSE PRINT "no:    ";
50    PRINT I;" "; J;
60    K = K + 1
70    IF K = 3 THEN PRINT: K = 0
80    NEXT I
90 END
```

```
yes:   1    1          yes:   2    2          yes:   3    3
yes:   4    4          yes:   5    5          yes:   6    6
yes:   7    7          yes:   8    8          yes:   9    9
yes:   10   10         yes:   11   11         yes:   12   12
yes:   13   13         yes:   14   14         yes:   15   15
yes:   16   16         yes:   17   17         yes:   18   18
yes:   19   19         yes:   20   20         yes:   21   21
yes:   22   22         yes:   23   23         yes:   24   24
yes:   25   25         yes:   26   26         yes:   27   27
yes:   28   28         yes:   29   29         yes:   30   30
yes:   31   31         yes:   32   32         yes:   33   33
yes:   34   34         yes:   35   35         yes:   36   36
yes:   37   37         yes:   38   38         yes:   39   39
yes:   40   40         yes:   41   41         yes:   42   42
yes:   43   43         yes:   44   44         yes:   45   45
yes:   46   46         yes:   47   47         yes:   48   48
yes:   49   49         yes:   50   50         yes:   51   51
yes:   52   52         yes:   53   53         yes:   54   54
yes:   55   55         yes:   56   56         yes:   57   57
yes:   58   58         yes:   59   59         yes:   60   60
yes:   61   61         yes:   62   62         yes:   63   63
yes:   64   64         yes:   65   65         yes:   66   66
yes:   67   67         yes:   68   68         yes:   69   69
yes:   70   70         yes:   71   71         yes:   72   72
yes:   73   73         yes:   74   74         yes:   75   75
yes:   76   76         yes:   77   77         yes:   78   78
yes:   79   79         yes:   80   80         yes:   81   81
yes:   82   82         yes:   83   83         yes:   84   84
yes:   85   85         yes:   86   86         yes:   87   87
yes:   88   88         yes:   89   89         yes:   90   90
yes:   91   91         yes:   92   92         yes:   93   93
yes:   94   93.99999 yes:   95   95         yes:   96   96
yes:   97   96.99999 yes:   98   98         yes:   99   99
yes:   100   100
```

FIGURE 3.2-13: This code shows how the precision problem can be overcome when comparing floating point values. The condition specified in line 40 requires that the two values only be within 0.001 of each other in order to be considered "equal."

not use 0.0001 for the tolerance level because we are approaching the limits of precision of BASIC, namely seven digits (two digits to the left of the decimal and five digits to the right).

Similar precautions must be used when dealing with *any* comparison of two floating point numbers, not just when testing for strict equality. This is because equality is always implictly being tested during a comparison. For instance, the statement

```
IF A < B THEN . . .
```

tests for equality, since the comparison will be *false* if A is equal to B.

Loops

The simplest and most commonly used loop is the FOR loop, which successively increments or decrements a loop control variable by a fixed amount. It is the mainstay of most BASIC programs. Its format and syntax are fairly simple, and most people can grasp its purpose quite easily. Yet, as simple as this construct appears to be, it is still a source of innumerable errors in programs. This is because the complexity of FOR loops is very deceptive. In addition, there has been an uncountable number of authors who have promoted very dangerous techniques for using FOR loops. It is an indication of just how devious these contructs are when experienced authors are not familiar with the pitfalls.

Branching and Loops

It is a well-known fact that a program should never attempt to branch into the middle of a loop of any kind. If branching must be done, it should always go to the loop statement itself, i.e., to the FOR, WHILE, or REPEAT statement. What is not generally recognized, however, is that it is very dangerous to branch *out of* a loop.

Figure 3.2-14 gives an example. This technique is widely recommended for inputting to an unknown number of values. The loop is effectively infinite (i.e., will execute more times than is practical if run to completion). The exit mechanism is the IF statement in line 260. The FOR statement itself is used simply as a convenient construct for creating a loop.

There are two problems with this technique. First, it grossly violates rules of structured programming by introducing a block with

```
210 PRINT "When done, enter 0"
220 PRINT
230 TOTAL = 0
240 FOR I = 1 TO 10000
250   INPUT "Enter a number: ",X
260   IF X = 0 GOTO 290
270   TOTAL = TOTAL + X
280 NEXT I
290 AVG = TOTAL / (I - 1)
300 PRINT AVG
```

FIGURE 3.2-14: A poor method of inputting an unknown-length list of values.

two exits. The first exit is the FOR statement's normal exit when the loop control variable exceeds its maximum value. The second exit is created by line 260.

But as has been said elsewhere, rules should never be adhered to so strictly that they become oppressive. Bending a rule slightly in order to gain a more convenient mechanism is often an acceptable compromise. However, a more devious problem may occur with some languages. In this case, a version of BASIC on some of the Radio Shack microcomputers produces an error when the next loop following the one that was jumped out of is executed. This error may be considered a bug in the BASIC interpreter of the Radio Shack computers, but the fact is that some versions of languages cannot handle such a branch.

Another reason for avoiding this technique is that the value of the loop control variable cannot always be depended upon to be what you think it should be once you exit the loop. Consider the code in Figure 3.2-15. What will the last value of I be when printed in line 40? Unfortunately, it depends entirely on which version of BASIC you are using. Some versions will print out 11, some 9, and still others 10! While you may find out how your current version of BASIC works, you can't depend on any other version to work the

FIGURE 3.2-15: Different versions of BASIC execute this code in different ways.

```
10 FOR I = 1 TO 10
20  PRINT I
30 NEXT I
40 PRINT I
```

same way. In addition, in languages such as Pascal, such a reference to a loop control variable outside of the loop produces an error.

If mechanisms such as the one shown in Figure 3.2-14 are tempting to use, simply do not use the FOR loop. Substitute a WHILE or REPEAT loop. However, remember to adhere to the principles of structured programming by retaining only a single exit from the loop. If the principles outlined in previous chapters for designing programs are followed, this situation should never occur, since there is no way to represent such a mechanism in the pseudo-code constructs presented. A later section in this chapter, Translating to Code, will discuss how these structured constructs can be translated into a GOTO-oriented language.

Loop Control Variables

Another technique to perform the same function as above is shown in Figure 3.2-16. This has also been suggested by a number of authors as a clever way to enter a variable number of values. In this case, when 0 is entered, the value of the loop control variable (I) is set to be greater than the maximum value specified by the FOR statement. Branching to the NEXT statement increments I, which is now obviously beyond its maximum value. As a result, the loop is exited normally.

This is, perhaps, in some ways to be preferred over the previous method discussed, since it preserves the structured programming principle of "one in–one out." However, not all versions of BASIC (or other languages) let you change the value of the loop control variable while inside the loop.

FIGURE 3.2-16: Another poor method for entering an unknown-length list of values.

```
210 COUNT = 0
220 PRINT "When done, enter 0"
230 PRINT
240 TOTAL = 0
250 FOR I = 1 TO 1000
260   INPUT "Enter a value: ",X
270   IF X = 0 THEN I = 1001: GOTO 300
280   TOTAL = TOTAL + X
290   COUNT = COUNT + 1
300 NEXT I
310 AVG = TOTAL / COUNT
320 PRINT AVG
```

```
10 FOR I = 1 TO 10
20   PRINT I
30   I = I - 1
40 NEXT I
```

FIGURE 3.2-17: This code can produce an infinite loop in many versions of BASIC.

The reason is that the incrementing or decrementing of the control variable is supposed to be automatic. This ensures that any changing of the value of the control variable is done correctly. Otherwise, problems such as that shown in Figure 3.2-17 can occur. In this example we have an infinite loop, since the variable I is decremented by the same amount that it is incremented by in the NEXT statement.

Even if the version of BASIC you use does let you change the value of the loop control variable, I strongly suggest you avoid this technique in order to eliminate the possibility of winding up with an infinite loop. Again, if you follow the techniques suggested in this book, this situation should never arise.

An equally troublesome problem is what happens when you change the value of any of the other control variables in the FOR statement. Recall that the general form of a FOR statement in BASIC is

FOR i = x TO y STEP z

where i is the loop control variable, x is an expression that, when evaluated, gives the initial value for i, y is an expression giving the maximum value that i can obtain, and z is the amount by which i is incremented each time the NEXT statement is executed. Since x, y, and z are expressions, they can contain variables themselves.

Figure 3.2-18 shows an example where the meaning of the code is obscure. How many times will I be printed, ten or seven? This depends on the language. Most will allow you to change a variable

FIGURE 3.2-18: This code shows how modifying a loop control variable can create an ambiguous logic.

```
10 Y = 10
20 FOR I = 1 TO Y
30   Y = 7
40   PRINT I
50 NEXT I
```

used in any of the expressions of the FOR statement without changing the number of times the loop will execute. This is because the expressions are evaluated only once, the first time the loop is entered. In other words, once the expressions *x*, *y*, and *z* have been evaluated, their values will not be changed by changing the value of variables used in these expressions while inside the loop. However, a situation such as that shown in Figure 3.2-18 is confusing at best, and should definitely be avoided.

WHILE vs. REPEAT

The final concern is whether the FOR loop in the language you are using is implemented like a WHILE loop, where the test is performed immediately, or like a REPEAT-UNTIL loop, where the test is not performed until the bottom of the loop. Recall that we specifically defined the FOR statement in the pseudo-code to be based on the WHILE loop. Pascal follows this specification explicitly.

Unfortunately, BASIC again does not follow any one pattern. Some versions implement it one way, some the other. Most appear to implement the FOR-NEXT loop as a WHILE loop. Figure 3.2-19 shows a simple program that will test your version of BASIC to determine which method it uses. If this code prints out the word "repeat," then your version of BASIC implements the FOR-NEXT loop as a REPEAT-UNTIL loop. If only the word "end" is printed, then FOR-NEXT is implemented as a WHILE loop.

Again, however, it is probably safest not to assume that any version of BASIC you are going to run a program on will implement FOR-NEXT in a particular way. Figure 3.2-20 shows how you can include code that will guarantee the FOR-NEXT loop is executed as a WHILE loop, even if the version of BASIC you are using has implemented it as a REPEAT-UNTIL loop. Unfortunately, there is no way to turn a FOR-NEXT loop that is implemented as a WHILE loop into a REPEAT-UNTIL version. In such a case, the REPEAT

FIGURE 3.2-19: This code can be used to determine whether the FOR loop of a version of BASIC acts as a WHILE loop or a REPEAT loop.

```
10 FOR I = 1 TO 0
20   PRINT "repeat"
30 NEXT I
40 PRINT "end"
```

```
110 IF X > Y GOTO 280
120 FOR I = X TO Y
      .
      .
      .
270 NEXT I
280 ...
```

FIGURE 3.2-20: **An example of how a simple IF statement (as in line 110) can be added to ensure that the FOR loop is treated as a WHILE loop instead of as a REPEAT loop.**

loop will have to be constructed in another way, as shown in the following section.

Translating to Code

If you are going to use Pascal or another structured language, then translating the design into code is fairly straightforward, since the control structures used to express the algorithm are available in the language. This is true to a certain extent with the more recent versions of BASIC, including the Microsoft version. These versions often include the IF-THEN-ELSE, WHILE, and REPEAT-UNTIL statements that form the skeleton of the structured languages.

Unfortunately, most versions of BASIC have another limitation that makes using the constructs difficult. The maximum length of any one statement in Microsoft BASIC, for instance, is 255 characters. This isn't a problem for the WHILE loop, since there is a second statement (WEND) which delimits the end of the loop, much like the NEXT statement does in the case of FOR. But for the IF statement, this limit can cause considerable difficulties.

Another problem with the IF-THEN-ELSE of BASIC is that it creates some very odd looking, and therefore potentially misleading, code. Consider the pseudo-code shown in Figure 3.2-21. Figure 3.2-22 shows how this pseudo-code would be translated into BASIC. Note the ELSE followed by another ELSE. This is absolutely necessary to ensure that the statements LET A = A − 1 and GOSUB 910 are performed when A >= 5, not when Q <= A. Each ELSE is applied to the closest IF statement that has not been matched with another ELSE.

In addition, the REPEAT-UNTIL and CASE constructs are not often implemented in BASIC. As a result, these control structures

```
IF A < 5 THEN
BEGIN
  increment A;
  set Q to Z * 47;
  IF Q > A THEN
  BEGIN
    decrement Q;
    Print-Out-Routine
  END
END
ELSE
BEGIN
  decrement A;
  Print-Out-Routine
END;
```

FIGURE 3.2-21: Later figures show how this pseudo-code can be translated into BASIC.

need to be simulated using other statements. We will discuss how to simulate all of these control structures using simple branching statements. The one caveat is that you must be cautious not to corrupt these constructs when you change or maintain the program. As will be pointed out in the next section, you should use comments to alert anyone reading the code as to exactly where these constructs are being simulated.

IF-THEN-ELSE

Let's begin with an example of simulating a simple IF-THEN statement. If we consider the general form of such a statement,

IF *condition* THEN *true-clause*

FIGURE 3.2-22: An example of how the algorithm shown in Figure 3.2-21 might be implemented in BASIC.

```
170 IF A < 5 THEN
      A = A + 1:
      Q = Z * 47:
      IF Q > A THEN
        Q = Q - 1:
        GOSUB 910                    'Print-Out-Routine
      ELSE
    ELSE
      A = A - 1:
      GOSUB 910                      'Print-Out-Routine
```

we recognize that we wish to execute the statements in the *true-clause* whenever the *condition* tests to be true. We can construct a functionally equivalent series of statements using the IF-GOTO form of the IF statement in the following manner:

```
100 IF NOT condition GOTO 170
110 true-clause
    . . .
170 . . . (next statement)
```

In this way, we branch around the statements of the *true-clause* whenever the condition is true. The reverse logic (NOT *condition*) of the IF statement causes a branch whenever its tested condition is true. When NOT *condition* is true, *condition* must be false.

For example, consider:

```
100 IF A = 5 THEN C = 12: PRINT A*C
. . . (next statement)
```

This can be simulated as:

```
100 IF A <> 5 GOTO 130
110 C = 12
120 PRINT A*C
130 . . . (next statement)
```

An IF-THEN-ELSE construct is a little trickier to represent. In order to preserve the order of the clauses in the code, reverse logic must again be employed. In general,

```
100 IF condition THEN true-clause ELSE false-clause
. . . (next statement)
```

would be converted to

```
100 IF NOT condition GOTO 190
110 true-clause
    . . .
180 GOTO 260
190 false-clause
    . . .
260 . . . (next statement)
```

Line 180 is necessary in order to branch around the *false-clause* statements when the *true-clause* has been executed. Figure 3.2-23 shows how the pseudo-code of Figure 3.2-21 can be converted to code using this method. Note that this does introduce some inefficiencies, such as the GOTO of line 200 branching to another GOTO. This should not be avoided, since it preserves the structured nature of the code. Another inefficiency here is the repeating of the code that decrements A and calls the *Print-Out-Routine*. Again, this should not be eliminated for the sake of efficiency, since such action would destroy the structure of the code.

While

Even though many versions of BASIC now include a WHILE loop, let's briefly look at how one can be simulated from simpler constructs. The flowchart form of the WHILE loop gives the clues we need for how to proceed. Figure 3.2-24 shows a WHILE loop in pseudo-code (a) and a flowchart (b) that can be drawn for it. The diamond shape in the flowchart indicates a branch, which can be performed with a simple IF-GOTO statement. This flowchart could then be used to generate the BASIC code given in part (c) of the figure. Note that, once again, negative logic had to be used in the condition of the IF statement.

Repeat-Until

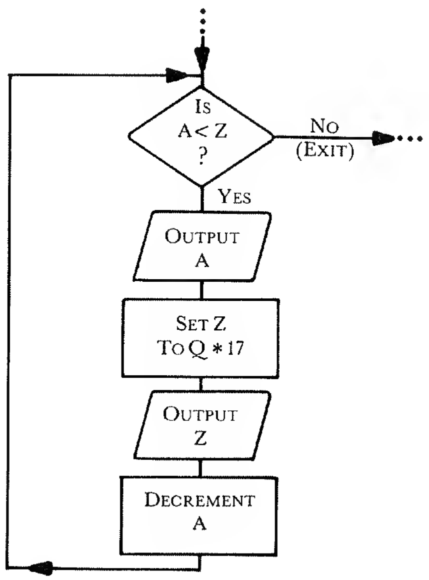
The reason the REPEAT-UNTIL construct is not included in BASIC is probably that it is so simple to construct with just an IF-GOTO statement. Consider the loop defined in Figure 3.2-25a. This can

FIGURE 3.2-23: The algorithm of Figure 3.2-21 implemented without the formal IF-THEN-ELSE mechanism. Lines 180 through 220 make up the THEN clause, while lines 240 and 250 make up the ELSE clause of the main IF statement. Line 260 is the next statement.

```
170 IF A >= 5 GOTO 240
180   A = A + 1
190   Q = Z * 47
200   IF Q <= A GOTO 230
210     Q = Q - 1
220     GOSUB 910                   'Print-Out-Routine
230 GOTO 260
240 A = A - 1
250 GOSUB 910                       'Print-Out-Routine
```

```
WHILE (A < Z) DO
BEGIN
  output A;
  set Z to Q * 17;
  output Z;
  decrement A
END;
```

a

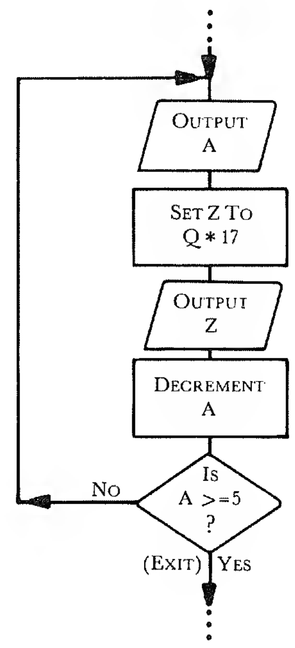


b

```
240 IF A >= Z GOTO 300
250   PRINT A
260   Z = Q * 17
270   PRINT Z
280   A = A - 1
290 GOTO 240
300 ... (next statement)
```

c

FIGURE 3.2-24a, b, and c: The pseudo-code in (a) can be translated in the flowchart shown in (b). This can then be easily translated in the BASIC code shown in (c).



a

```
360   PRINT A
370   Z = Q * 17
380   PRINT Z
390   A = A - 1
400 IF A < Z GOTO 360
410 ... (next statement)
```

b

FIGURE 3.2-25a and b: The flowchart in (a) shows a REPEAT-UNTIL loop. This can be translated into the BASIC code shown in (b).

easily be coded, as shown in Figure 3.2-25b. Note, again, the reverse logic used in statement 400.

Case

The CASE is an imposing statement when viewed in pseudo-code form. Consider the example in Figure 3.2-26a. It is not immediately obvious how this might be translated into code without a CASE statement.

The clue comes easily from the flowchart, as shown in Figure 3.2-26b. Each case can be considered an individual test of the same

```
CASE today OF

Monday:          BEGIN
                   Q := 17;
                   Print-Out-Routine
                 END;

Tuesday..Thursday:  Z := Q * 5;

Friday:          BEGIN
                   Q := 47;
                   Z := A / 6;
                   Print-Out-Routine
                 END;

OTHERWISE:       Q := 0

ENDCASE;
```

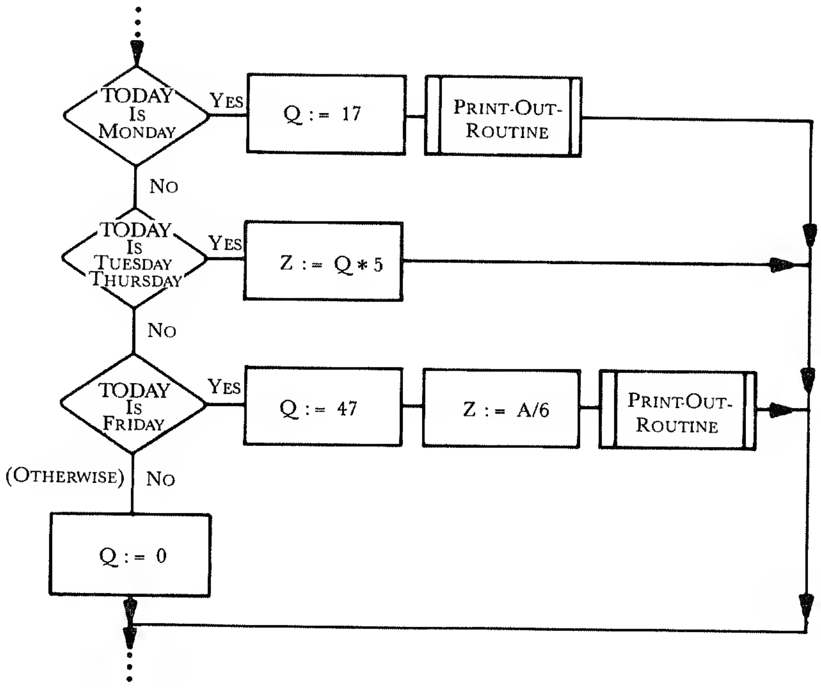


FIGURE 3.2-26a and b: Part (a), top, shows an example of a CASE statement in pseudo-code. There is an obvious problem when trying to translate this statement into a programming language that does not contain a CASE statement. Part (b), bottom, shows how this pseudo-code can be drawn in a flowchart. Part (c), top right, shows how this flowchart can then be used to create appropriate code in BASIC, which typically does not contain a CASE statement.

```
730 IF TODAY$ = "MONDAY" THEN
       Q = 17:
       GOSUB 1110:                    'Print-Out-Routine
       GOTO 770
740 IF (TODAY$ = "TUESDAY") OR (TODAY$ = "WEDNESDAY")
       OR (TODAY$ = "THURSDAY") THEN
         Z = Q * 5:
         GOTO 770
750 IF TODAY$ = "FRIDAY" THEN
       Q = 47:
       Z = A / 6:
       GOSUB 1110:                    'Print-Out-Routine
       GOTO 770
760 Q = 0
770 ... (next statement)
```

FIGURE 3.2-26c

variable. In this example, the case selector is the variable TODAY, which can be tested against the values given in each case. When a match is made, the statements associated with that case are executed.

Each case can, therefore, be treated as an individual IF statement. The flowchart could be coded in BASIC as shown in Figure 3.2-26c.

If the IF-THEN construct isn't going to allow you to put enough statements on the same line number (because of the 255 character limit), you'll have to simulate this with an IF-GOTO construction as discussed previously.

3·3 PROGRAM STYLE

In this section, we will discuss how to make the general format of a program more readable. As always, any improvement in readability of the code will increase the programmer's ability to maintain the program properly. It will speed up his or her searches for specific code, and aid him or her in understanding that code.

There are few functional requirements for the format of a program, although each language requires the general outline of a program to follow certain forms. For instance, in Pascal, a PROGRAM statement must be the first statement of every program. Indeed, Pascal in many ways can be said to have a very rigid format requirement, while BASIC has a very free format.

Each programmer has a tendency to develop his or her own style. This is perhaps as it should be, since programming is a creative activity. However, when a programmer follows his own style, it may make his code more difficult for another programmer to read. This is especially true when the programmer does not follow one style consistently, but changes style throughout a program. This occurs typically because the programmer hasn't quite settled on one style, or because he simply did not take the time to be consistent.

Adding style to a program is time-consuming and is considered by some, along with general documentation, to be unproductive. However, it has been shown in research that a well-styled program is easier to maintain, so it makes sense to style your programs. Maintainability has been the ultimate goal of nearly everything covered in this book so far.

In the discussions that follow, the formats suggested are based upon many years of professional programming experience. They have worked for me, so I assume that they will work for you. However, they are not hard and fast rules that must be followed. Feel free to add to these guidelines any ideas that will make the code more readable for you. Be certain, however, that any format you try is consistently applied, at least within a single program. Also, keep a notebook of any styles that you intend to use, complete with an example. This makes it easier to apply these styles consistently with each new project.

General Format

First, let's look at some ideas for defining a program's format in general terms. This is already done to a large extent in Pascal, where the language forces us to place all declarations in a specific order. In BASIC, however, there are few such restrictions. As a result, it will be necessary to define our own format for BASIC.

BASIC Program Format

A program can be thought of as being composed of two distinct parts, declarations and executable code. We might wish to include a third part, comments, that have no effect on the outcome of a program's execution. By identifying these three parts, we can more clearly define their respective roles in the format of the program.

PROGRAM HEADER Comments are used strictly to give additional information to the programmer as he or she is reading the code. They are not available to a user of the program. Typically, comments will be scattered throughout the program to give a running commentary of the action. It is helpful, however, to begin each program with a special series of comments called a **program header,** which provides general information about the nature of the program. The contents of the program header are discussed in detail in section 6.2, Internal Documentation.

DECLARATIONS The next section of the program should consist of any declarations that must be made. These generally appear as the first statements in a program because their job is to set up variables and workspace for the program to use. Figure 3.3-1 gives a list of BASIC statements that would fall into this category. They are listed in an order that usually should be followed in the program.

In Microsoft BASIC, a special statement, %INCLUDE, can be used anywhere in the program to copy statements from a file into the program. %INCLUDE could be used to copy a predefined set of declarations into the program. This might change the order of the resulting list of declarations somewhat, but as long as the syntactic rules concerning the order of declarations for the version of BASIC being used are followed, there is no harm in this.

Note that DATA statements have been included in the list. Such statements are not part of the executable code, since they do nothing by themselves. They depend on the READ statement to activate them. I feel it is much more understandable to lump all DATA statements together in one location in the program, rather than pair the data statements with individual READ statements throughout the program, as promoted by many authors. This is because BASIC

FIGURE 3.3-1: A list of typical BASIC commands used to set up variables and workspace for a program to use. These commands should usually be included in the declaration part of a program.

```
OPTION BASE
DIM
COMMON
MERGE, CHAIN
DEF
FIELD
DATA
```

treats all READ statements as contiguous anyway, no matter where they are located in the program. Placing all DATA statements together emphasizes this fact. In addition, it removes them from the main program, where they invariably clutter the code, making it more difficult to read.

USER INSTRUCTIONS If this is to be an interactive program, i.e., one which will output messages to a user and will expect input from the user, it is usually a good idea to output a general message to the user at the very beginning of the program. This message could include a short description of the purpose of the program, letting the user know that the proper program has indeed been executed.

In addition, this section should output the beginning instructions to the user on how the program is used. In more sophisticated systems, this could include multiple levels of help facilities and start-up instructions, for a range of users from novice to expert. At the very least, this output should describe, in a general way, what is expected of the user.

INITIALIZATION This section of the program gives initial values to variables that will be used in the program. In addition, it executes statements that set up other types of initialization.

Figure 3.3-2 gives a list of statements commonly included in this section. Once again, these statements are specified in a recommended order due to certain constraints on the way BASIC functions.

OPEN statements are included here for files that are going to be used throughout the program. This is not a strict requirement, however, since there are many occasions when a file must be opened and closed numerous times throughout the program. In this case, the OPEN statement would appear wherever it is appropriate in the program.

The CLEAR statement resets all variables to zero, all strings to

FIGURE 3.3-2: A list of BASIC statements that are used for initializations of variables, files, etc.

```
ON ERROR
CLEAR
RANDOMIZE
OPEN
```

null, and closes all files. Even though most languages perform such initialization automatically when a program is first entered, I strongly recommend using this statement explicitly in every program. Its effect on efficiency is minute, and such an initialization could save you great headaches later on during debugging.

Although this seems to cover all types of initialization for variables, there are usually other initial values that must be set up. Therefore, follow this section with any LET statements needed for that purpose.

Initializations are typically meant to be executed only once. However, certain programs may require re-executing at least a portion of the initializations. In such a case, make certain that nothing gets reset unless it is supposed to. On the other hand, make certain that *everything* that needs to be reset is!

MAIN PROGRAM The main execution of the program comes next. The content of this section will obviously vary greatly from program to program. However, you will be surprised at how many programs follow a simple pattern of input, processing, and output. You should be able to easily identify such sections in your programs if you have followed the guidelines discussed previously.

SUBROUTINES AND FUNCTIONS Finally, it is helpful in many ways to group all subroutines and user-defined functions together in one place. The most out-of-the-way location is at the end of the program. This keeps the necessity of branching around subroutines to its absolute minimum. It requires, however, that the subroutine section be preceded by a statement that will keep normal execution from "falling into" the subroutines. Such a statement would be either STOP or a GOTO the END statement.

END Programs should always have a defined end. Although BASIC does not usually require an END statement, I strongly recommend it. This keeps with the structured principles previously outlined and gives a place to GOTO from anyplace in the program to halt execution. This is preferable to numerous STOP statements because it preserves the "one out" philosophy of modules. I generally number the END statement with all 9s, making it very obvious in the program whenever I'm branching to the END.

SUMMARY The following gives a summary of the general format of a well-styled BASIC program:

1. Program header
2. Declarations, including DATA statements
3. Start-up instructions to the user
4. Initialization
5. Main program
6. GOTO end
7. Subroutines
8. END

Pascal Program Format

In Pascal, the general layout of a program is strictly defined and cannot be altered. Since Pascal requires all declarations, including procedures, to follow this strict order, the only place for any personal styling is in the main program block itself.

The pattern of executable statements can vary greatly from program to program, making the development of a single, well-defined layout difficult. However, since programs follow similar patterns no matter what language they have been written in, the same general format for the executable statements portion of the program will apply to Pascal as well as to BASIC.

The only exception to this is in the placement of the subroutines (called **procedures** in Pascal). Pascal considers these to be declarations and requires that they be specified in the declarations section of the program.

This leaves only the instructions, initializations, and main program sections, which should follow the same order as before. Obviously the details of such sections will be different.

In the case of initializations, however, the types of actions still apply. Any error initialization should be performed first. Then, setting up variable and file initializations should be handled. There are typically no special keywords to perform many of these functions in Pascal. The reference manual for the compiler you are using will have to be consulted for details of how to accomplish these actions.

Indentation

One method of making a program listing more readable is to use an indentation technique for each block of code. This makes each block stand out more and helps to highlight the scope of the block.

Indentation can be implemented simply by indenting, say, two spaces each time a new block is encountered. When the block ends, remove the indentation for that block. This indentation follows the same form as that shown in section 2.2, The Use of Pseudo-code.

Figure 3.3-3 gives an example of indentation of BASIC. Each time a new level of embeddedness of control structures is encountered, additional indentation is made. Note that pairs of FOR and NEXT statements, WHILE and WEND statements, and IF and ELSE (when there is one) statements are on the same level of indentation. This makes identifying the scope of these constructs more obvious.

Figure 3.3-4 shows how this might be handled in Pascal. We must use the BEGIN-END construct for denoting blocks in Pascal. Note that they are aligned with the control structure to which they apply. Also note the use of comments (identified by {} in Pascal) to help identify which END goes with which BEGIN.

There are programs, often called **pretty printers,** that will do this type of indentation automatically, but by whatever means it is accomplished, indentation is worth the effort.

FIGURE 3.3-3: An example of using indentation in a BASIC program to indicate the scope of control structures.

```
180 FOR I = 1 TO 25
190   IF I = 16 THEN
        PRINT:
        Z = 47
      ELSE
        IF A$ = "YES" THEN
          GOSUB 1110:
          B$ = "YES":
          PRINT I
200   WHILE A$ = "YES"
210     X = 0
220     Z = A * I + C(I)
230     FOR J = I TO K
240       PRINT "*";
250       IF J = INT((K-I)/4) THEN
            PRINT
260     NEXT J
270     IF Z = 47 THEN
          Z = 0:
          INPUT A$
        ELSE
          A$ = "NO"
280   WEND
290   PRINT
300 NEXT I
```

```
for I := 1 to 25 do
begin
  if 1 = 16 then
  begin
    writeln;
    z := 47
  end                          { if }
  else
    if ASTRING = 'YES' then
    begin
      Out-Routine;
      writeln(I)
    end;                       { if, also else }
  while ASTRING = 'YES' do
  begin
    X := 0;
    Z := A * I + C[I];
    for J := I to K do
    begin
      write('*');
      if J = trunc((K-I)/4) then
        writeln
    end;                       { for }
    if Z = 47 then
    begin
      Z := 0;
      readln(ASTRING)
    end                        { if }
    else
      ASTRING := 'no'
  end;                         { while }
  writeln
end;                           { for }
```

FIGURE 3.3-4: Using indentation in a Pascal program to indicate the scope of control structures. The comments are added to increase the readability by identifying an *end* with the statement to which it applies.

Comments

I am a firm believer in the usefulness of documentation. Comments within a program can make the code much more readable and understandable. The specifics of such comments are covered in section 6.2, Internal Documentation. However, here we will concern ourselves with the form that comments take.

Block Separators

It is amazing how separating lines of code with blank lines can make it more readable. Such separators between blocks enhance the modularity of the code. These can be added simply by placing blank comments between lines. In most modern compiled languages, blank lines can be added with the editor used to create the program code. In a language like BASIC, specific blank comments must be added. This uses up line numbers in BASIC but is still easily accomplished.

Most compiled languages also include commands for causing special spacings in a listing. The most often used one causes a **page break,** i.e., the listing skips to the top of a new page before continuing. Such commands do not affect the execution of the program or the program output. They are used by the compiler only to generate a listing of the program. The page break can be used to begin modules on a new page.

Another useful technique is to print entire lines of a special character to indicate a particular type of block. It might be helpful, for instance, to use a line of asterisks in between each subroutine in the program. This makes the subroutines easier to locate.

Control Structures

If you have built your own control structures using the IF-GOTO statement, as discussed in section 3.2, Implementation Guidelines, comments should be used to identify what the constructs are as well as their scope. This is important so that such constructs are easily recognized when you maintain the program the next time. Figure 3.3-5 gives an example of such comments for a BASIC program.

Detail Line Comments

You should not feel compelled to comment every line of code. In fact, if you have done so, you have either wasted your time or have not written the code properly. Such **detail line comments** may be necessary when using assembly language, but should not be required when using a high-level language.

When comments are used, they can generally be added either by including entire lines of comments, or by adding a comment on the same line as code. The former approach should usually be taken when commenting on an entire block of code. The latter method is sometimes necessary to describe a particularly tricky section of the program. In this case, I typically try to make all of my comments start in the same column of the line—for instance, in column 60. In addition, I try not to let my code run past this column. This makes the comments stand out more without obscuring the code. Remember that making the code readable is what counts.

```
190 FOR I = 1 TO 25
195 '
200   IF I <> 16 GOTO 250
210     PRINT                       ' if I = 16, then
220     Z = 47
240     GOTO 300                    ' end then (I = 16)
245 '
250     IF A$ <> "YES" GOTO 300     ' else, if I <> 16
260       GOSUB 1110                ' if A$ = "YES", then
270       B$ = "YES"
280       PRINT I                   ' end then (A$ = "YES")
290                                 ' end else (I = 16)
300   IF A$ <> "YES" GOTO 450       ' while A$ = "YES" do
310     X = 0
320     Z = A * I + C(I)
330     J = I
335 '
340       PRINT "*";                ' repeat
350       IF J = INT((K-I)/4) THEN PRINT
360       J = J + 1
370     IF J < K GOTO 340           ' until J >= K
375 '
380     IF Z <> 47 GOTO 420
390       Z = 0                     ' if Z = 47, then
400       INPUT A$
410       GOTO 450                  ' end then (Z = 47)
415 '
420       A$ = "NO"                 ' else, if Z <> 47
430                                 ' end else (Z = 47)
440                                 ' end while (A$ = "YES")
450   PRINT
460 NEXT I
```

FIGURE 3.3-5: **An example of how comments can be added to identify what control structures are being used, and their scope, in a BASIC program segment. Note, too, the use of indentation to aid in identifying the scope of the structures.**

Figures 3.3-6 and 3.3-7 give examples of these techniques in BASIC and Pascal, respectively. Note in the BASIC example the use of the apostrophe instead of the keyword REM to indicate a comment. Having REM all over the place in a listing tends to clutter it.

3·4 EFFECTIVE USE OF SUBROUTINES

Like many of the techniques we have discussed thus far, subroutines can either make our lives as programmers miserable, or make them simpler by providing a tool for constructing well-ordered programs. The first instinct that must be modified in the programmer is that

```
180 PRINT
190 PRINT
195                                                 ' repeat
200 INPUT "Enter any integer number: ", NUMBER
210                                      ' verify it's integer
220   IF NUMBER - INT(NUMBER) <> 0 THEN
          PRINT "Number is not an integer --reenter":
          GOTO 200                       'until valid
230 '
240 ' reverse the number
250 '
260 X = NUMBER
270 Z = 0
280 IF X <= 0 GOTO 350                            'while x > 0 do
290   W = INT(X / 10)
300   Y = X - W * 10       'y is right-most digit of x
310   Z = Z * 10 + Y       'put y digit on right side of z
320   X = W                'w is x without its right-most digit
330 GOTO 280                                           'end while
340 '
350 PRINT
360 PRINT
370 PRINT "original", "reversed"
380 PRINT "number", "number"
390 PRINT "------", "------"
400 PRINT NUMBER, Z
410 PRINT
420 '
430 ' if the number that was input is the same as the
440 ' reversed number, then the number is a palindrome (i.e.
450 ' the same digits read right to left as read left to right
460 '
470 IF NUMBER = Z THEN
         PRINT "the number is a palindrome"
    ELSE PRINT "the number is not a palindrome"
```

FIGURE 3.3-6: An example of the use of detail line comments in a BASIC program.

which says the subroutine's primary raison d'etre is to eliminate duplicate code from a program. As we have already discussed, creating structured programs often *causes* the duplication of code. However, the overall effect of implementing modules as subroutines is generally very pleasing.

Modularity, Top-down, and Bottom-up Revisited

Subroutines will be the implementation of modules, sort of the flesh and blood manifestation of the spirit. The creation of subroutines will aid in the isolation of functions within the system. Therefore, should anything need to be changed with regard to a particular

```
program palindrome(input,output);
var number, w, x, y, z: integer;
begin
  writeln;
  writeln;
  writeln('Enter any integer number: ');
  readln(number);

{ reverse the number }

  x := number;              { initializations to get started }
  z := 0

  while x > 0 do
  begin
    w := x div 10;          { div is integer division }
    y := x - w * 10;        { y is right-most digit of x }
    z := z * 10 + y;        { put y digit on right side of z }
    x := w               { w is x without its right most digit }
  end;

  writeln;
  writeln;
  writeln('original number: ',number);
  writeln('reversed number: ',z);
  writeln;

{ if the number that was input is the same as the reversed
  number, then the number is a palindrome (i.e. the same
  digits read right to left as read left to right          }

  if number = z
  then
    writeln('the number is a palindrome')
  else
    writeln('the number is not a palindrome')

end.
```

FIGURE 3.3-7: An example of line comments in a Pascal program. Blank lines can be added to a Pascal program without using an explicit comment.

function, the changes to the code are generally limited to a single or series of subroutines. This makes the maintenance of the system infinitely easier.

In addition, this isolation of functions helps in debugging the system. First, debugging can be done **incrementally,** on each module as it is created rather than on the system as a whole. More important, bugs are generally limited in their scope of "damage," i.e., how far the execution continues from the actual source of the error. Subroutines form a kind of natural boundary that is difficult for bugs to cross. Also, bugs can be more easily traced back to a subroutine than to a general area of code within the entire program. Often when a bug is isolated as being within a particular subroutine, that subroutine can be tested independently from the rest of the code of the system.

In implementing subroutines, we can again take either the top-down or bottom-up approach, depending on our own whims. In the bottom-up method, simply begin programming the lowest-level module first, successively building up levels until the topmost level is reached. Then return to any remaining module at the lowest possible level to begin implementing again. In addition, you can test each module as it is implemented, giving you the security that all code below the level you are currently working at is probably "bug free." This is discussed in detail in Chapter 5, Program Testing and Debugging.

Personally, I find the bottom-up approach more satisfying. First, I feel more comfortable knowing all the code beneath the level I'm working on when implementing a routine. Doing things top-down can occasionally lead to the situation of coming to the bottommost level and realizing that you forgot to do something at a higher level. Second, I find testing a bottom-up implementation less tedious than testing one developed top-down. It just seems easier to set up one set of input parameters for testing a series of subroutines than to set up output parameters in all subroutines for testing a top-down implementation.

Problems

It would be nice to think that using subroutines solved all our problems without creating any new ones. No such luck. There are always trade-offs in the computer field. In this case, using subroutines in certain programming languages can introduce new types of errors into your code. Protecting against the occurrence of such bugs is often tedious. However, the overall benefits of using subroutines to their best advantage can far outweigh any disadvantages.

A first problem is that many languages allow a programmer to code multiple entries to or exits from a subroutine. The most notorious of these languages is BASIC, mainly because it doesn't employ any special mechanism other than a RETURN statement for iden-

```
510 I = 0
515 INPUT X(I)
520 IF X(I) = 0 GOTO 550
525 I = I + 1
530 IF I <= 100 GOTO 515
540 RETURN
550 PRINT
560 PRINT
570 FOR J = 1 TO I
580   PRINT X(J)
590 NEXT J
600 RETURN
```

FIGURE 3.4-1: An example of the ambiguity of the BASIC subroutine mechanism. Does this code show one subroutine, or two? Even reading the code very carefully does not conclusively indicate which.

tifying a subroutine. Look at Figure 3.4-1. We see two RETURN statements in the code. This might lead us to think that there are two subroutines defined here. However, there could just as easily be a single subroutine, since BASIC allows us to have as many RETURNs as we wish. In addition, even if this were intended to be a single subroutine, there is nothing to prevent us from having a GOSUB 550, for instance, somewhere else in the program where we simply want to have the array X printed. There is no statement that identifies the *beginning* of a subroutine in BASIC.

Obviously, the use of multiple entry or exit points for a subroutine violates our "one in–one out" philosophy and should, therefore, be avoided like the plague. A later section on Guidelines for BASIC Subroutines discusses how to program around this problem.

A second deadly problem with subroutines is the use of **global variables.** We can define the **scope** of a variable as being anyplace in the program code where the value of that variable is known, i.e., can be used or modified. A global variable has a scope that encompasses all subroutines below the level on which it is declared. Figure 3.4-2 shows a Pascal program that declares three variables, *sum, i,* and *value,* at the level of the main program. These variables can also be used within the subroutine called *sumit,* making them global to that subroutine. In this example, the value of *sum* is calculated within the subroutine, and then used within the main routine to output the sum of the ten numbers read in.

```
program example(input,output);

var sum, i, value: integer;

{ the following is a subroutine }

procedure sumit;
begin
  sum := 0;
  for i := 1 to 10 do
  begin
    readln(value);
    sum := sum + value
  end
end;          { this is the end of the subroutine }

begin         { this is the beginning of the main program }
  sumit;      { call the subroutine }
  writeln(sum)
end.
```

FIGURE 3.4-2: An example of global variables in a Pascal program. All three variables (*sum, i,* and *value*) are used as global, meaning that they can be referenced (used) within both the main program and the procedure *sumit.*

Pascal Conventions

In Pascal, all variables must be **declared,** i.e., the Pascal compiler must be told explicitly what **type** a particular variable is. In the example of Figure 3.4-2, the variables *sum, i,* and *value* are all declared to be integers. Other types include *real* and *char,* which are roughly equivalent to floating point numbers and strings in BASIC. In addition, each Pascal procedure (subroutine) is declared by giving it a name. A call to that procedure is performed simply by using its name. Note how this is accomplished in the main program of the example.

In fact, in Pascal any variable declared at the main program level is global to any subroutine, unless certain steps are taken to prevent this. Figure 3.4-3 shows how variables can be made **local** to a particular portion of code, in this case the subroutine *sumit.* Because the variables *i* and *value* have now been declared at the *sumit* level, these

```
program example2(input,output);
var sum: integer;

{ this is the beginning of the subroutine }

procedure sumit:
var i, value: integer;
begin
  sum := 0;
  for i := 1 to 10 do
  begin
    readln(value);
    sum := sum + value
  end
end;          { this is the end of the subroutine }

begin         { this is the beginning of the main program}
  sumit;
  writeln(sum)
end.
```

FIGURE 3.4-3: An example of using local variables in a Pascal program. In this case, the variables *i* and *value* can only be referenced (used) within the procedure *sumit*. The variable *sum* is still used globally, however.

variables cannot be referenced in the main program as *sum* can. Therefore, they are local to the subroutine *sumit*. Their values cannot be accessed or modified anywhere else in the program except within the *sumit* procedure.

Using local variables is a way of limiting the scope of variables, isolating them so that they cannot be accidentally used or changed when we don't really want them to be. We call these "accidental" changes to a variable within a subroutine a **side effect** of the subroutine. The change of *sum* in the examples shown in Figures 3.4-2 and 3.4-3 is not a side effect since we intended for its value to change as a result of calling the *sumit* subroutine. In addition, we can limit the occurrence of side effects in Pascal by using local variables and parameters, which will be discussed in the next section.

In BASIC, we don't have as neat a mechanism as we do in Pascal for dealing with global and local variables. In fact, *all* variables are global in BASIC, which means that the value of a variable is known *everywhere in the program*. This is mainly because variables are not declared in BASIC. In other words, you can introduce a new variable name any place in the code. The only exceptions to this are arrays (declared with the DIM statement) and strings in some dialects of BASIC.

```
100 GOSUB 210
110 PRINT SUM
120 GOTO 999
130 SUM = 0
140 FOR I = 1 TO 10
150   INPUT VALUE
160   SUM = SUM + VALUE
170 NEXT I
299 RETURN
999 END
```

FIGURE 3.4-4: An example of a BASIC program using subroutines. In BASIC, *all* variables are considered to be global because of the nature of the language. This is one of the main weaknesses of BASIC. This is made obvious from the example in Figure 3.4-5.

Figure 3.4-4 shows an example in BASIC of the same program for adding ten numbers that we previously saw implemented in Pascal. The function is exactly the same as in Figure 3.4-2, where all variables were global. At line 110, in addition to the sum of the numbers, we could output the value of the *I* and *VALUE* variables if we wished. The variable *I* would have the last value of the index variable from the FOR statement in the subroutine, while *VALUE* would have the value of the last number that was entered. This all works perfectly well, and there are no noticeable side effects.

Now look at Figure 3.4-5. Here the main program uses *I* to hold the age of the user. However, notice what happens to *I* once it is inside the subroutine. There is a loop there that prints an entire row of asterisks. Unfortunately, the index variable is *I*. What happens to the age of the user when this subroutine is executed? It disappears because the location in memory we called *I* is initialized to 1, and then incremented past 70 within the FOR loop. Therefore, no matter what age the user gives in response to the prompt in line 110, the age printed out in line 140 will probably be 66!

If we were to look at just the subroutine, we wouldn't notice anything wrong with it. Likewise, looking at the main part of the program also does not reveal anything amiss. It isn't until we diligently trace the execution of the code that we discover that the value of *I* is inadvertently changed within the subroutine. This side effect causes an error to appear in the output. More seriously, such a side effect could create a fatal error, causing the program to crash. Unfortunately, there are no inherent mechanisms in BASIC to keep such side effects from happening.

```
100 INPUT "What year is this ";YR
110 INPUT "Enter your age: ",I
120 GOSUB 210
130 PRINT "You were born in "; YR-I
140 PRINT "You don't look a day over ";I - 5
150 GOTO 999
210 PRINT
220 FOR I = 1 TO 70
230   PRINT "*";
240 NEXT I
250 PRINT
260 PRINT
299 RETURN
999 END
```

FIGURE 3.4-5: An example of a side effect, where the execution of a subroutine messes up the value of a variable. In this case, the variable *I* is used as a loop control variable in the subroutine, while it is used to hold the age of the user in the main program.

Parameters

Back in the section on algorithms, we informally defined the notions of input and output parameters. To recap, an input parameter is any variable whose value is established before calling a subroutine, and which is subsequently used within that subroutine. An output parameter is any variable that we expect a called subroutine to change so that the new value can be used in the calling routine. A variable can be both an input and an output parameter if the value that the variable had upon entering the subroutine is used within the subroutine, and if the subroutine will also modify the value of the variable.

The main purpose of parameters is to limit (if not totally eliminate) side effects when calling a subroutine. The idea is that the only variables whose values are used within the subroutine are either input parameters or local variables, and that the only way for a subroutine to change a variable that is used outside that subroutine is for that variable to be an output parameter.

Many programming languages contain a formal mechanism for using parameters. For instance, the example in Figure 3.4-6 shows how Pascal handles parameters. Following the name of the procedure *sumit* is a list of the parameters that will either be passed to the procedure as an input, or passed back as an output. In this case we have two parameters, *numbr* as an input parameter, and *sum* as an

```
program example3(input,output);
var sum, numbr: integer;

procedure sumit(numbr: integer; var sum:integer);
var i, value: integer;
begin
  sum := 0;
  for i := 1 to numbr do
  begin
    readln(value);
    sum := sum + value
  end
end;                         { end of subroutine }

begin                        { begin the main program }
  readln(numbr);
  sumit(numbr,sum);
  writeln(sum)
end.
```

FIGURE 3.4-6: An example of using parameters in procedures in a Pascal program.

output parameter. The output parameters are easily identified because they must be preceded with a *var* designation. The parenthesized expression acts as the declaration for the variables that will be called *numbr* and *sum* in the subroutine. Any reference to these two names within the subroutine refers to the variables defined in this declaration.

So what about the *numbr* and *sum* that were declared in the main program? After all, a value for *numbr* that tells us how many numbers will be read and summed in the subroutine was actually read in the main program.

The names used in the main program refer to locations (addresses) within memory different from the locations of the variables with the same names in the procedure *sumit*. To help keep things straight, let's for the moment refer to the variable *numbr* that is declared in the main program as $numbr_m$, and to the variable *numbr* that is declared as a parameter in the procedure *sumit* as $numbr_p$. During the execution of this program, the following events take place:

When we are executing statements in the main program, such as *read*(*numbr*), the statements refer to the location in memory pointed to by the name $numbr_m$. Figure 3.4-7 shows that the only two variables we can reference in the main program are sum_m and $numbr_m$. No other variables have space set aside in memory for holding a value.

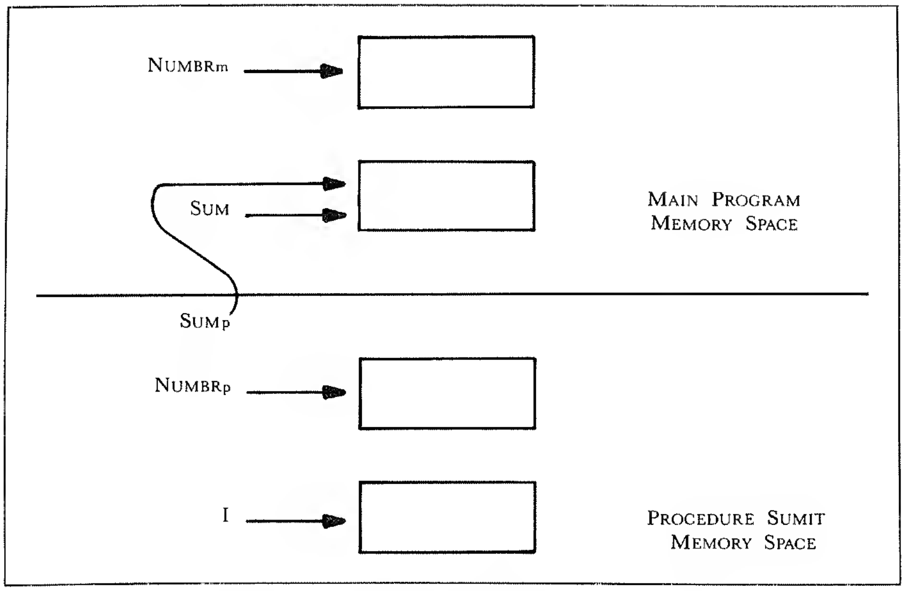


FIGURE 3.4-7: A graphical representation of the memory space that can be referenced when either the main program or *sumit* procedure is executing.

When the *sumit* procedure is called by the main program, the main program's execution is suspended, and we enter the subroutine's **environment.** This means that a different area of memory, set aside for use only by the procedure *sumit*, is now executing. When *sumit* finishes executing, we will return to the environment of the main program. Within the *sumit* environment, the only variables that we can reference are sum_p, $numbr_p$, and i. Figure 3.4-7 shows that only spaces set aside in memory at this time are for these variables. No other variables can be referenced, including $numbr_m$. As far as the procedure *sumit* is concerned, this variable does not exist.

At the time that the call from the main program to *sumit* is made, any variables that are used in the call statement itself (we call these the **actual parameters,** in this case $numbr_m$ and sum_m) have their values copied to the corresponding variable name (called the **formal parameter**) that is an input parameter. If, for instance, we had entered the value 5 for the number of values to be summed, that value would be copied from $numbr_m$ into $numbr_p$ when the call is made. Thus, the subroutine can use the value that it got from the caller *without directly referencing the same physical location in memory.* That's the key, since now we cannot inadvertently change the value of $numbr_m$ within the subroutine. Any reference to *numbr* within *sumit* refers to what we have called $numbr_p$, and not $numbr_m$.

However, output parameters are handled differently, thus the need for using the *var* tag in front of the output parameter declaration in the procedure statement. In this case, referencing *sum* in the procedure *sumit* really does directly reference the same physical memory location as the variable named *sum* in the main program. Therefore, any change to *sum* within *sumit* will change the value of *sum* in the main program.

So why not just use global variables instead of the formality of output parameters? Isn't the net effect the same? As you might expect, the answer is both yes and no. Using output parameters can help eliminate side effects by requiring all variables used within a subroutine to be either input parameters, output parameters, or local variables. It is this last requirement that is the mortar for the brick wall we have erected against side effects. Essentially, we have outlawed the use of global variables.

As it turns out, this is a great idea, and works very effectively when dealing with simple variables. However, the benefits of the entire system begin to break down when we attempt to pass complex data structures, such as arrays, as parameters. In this case, it is much more efficient to use global variables to refer to the structure than to actually pass it as a parameter. So global variables have their place in the scheme of things, too.

Guidelines for BASIC Subroutines

BASIC lacks any but the most rudimentary form of a subroutine. As mentioned earlier, the only special statement defined for subroutines, in addition to the calling GOSUB, is the RETURN. This in itself gives us very little help in using subroutines effectively. Therefore, it is up to us as clever programmers to come up with our own mechanisms to make using subroutines less hazardous. The techniques I will suggest may seem like the hard way of doing things, and undoubtedly they are occasionally just that, but they do bend the inhospitable world of BASIC subroutines into a form that can be safely used by any programmer.

Style

Dwyer and Critchfield [35] suggest a general format for BASIC subroutines that I find very useful. It involves little more than using comments and a simple numbering system to set up every subroutine into a common pattern. In this way, it is hoped that the subroutine will be less mysterious when it comes time to decipher it during debugging.

First, number the first statement of all subroutines with a line number that ends in 10. In this way, the beginning of the subroutine is obvious. In the subroutine shown in the program of Figure 3.4-8, the subroutine code begins on line 510. This idea also supports our "one in" philosophy, since now the only way into this subroutine should be through the subroutine "header."

Second, number the RETURN statement for a subroutine with the next highest number needed that ends with a 99. In the example, the end of the subroutine is line 599. This "wastes" a couple of line numbers that might have been used, but think of it more as allowing some room for the growth of the subroutine later. In addition, this numbering forces the RETURN statement to be the last physical line of the subroutine, again making it obvious which statements belong to each subroutine.

FIGURE 3.4-8: An example of how a BASIC program can be styled to enhance understanding. Note especially the use of special variables within the subroutine to simulate "local" variables.

```
100 INPUT "How many numbers "; NUMBR
105 '>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
110       'Prepare input parameter for routine 510
120 NUM5 = NUMBR
130 GOSUB 510                            'CALL SUMIT(SUM,NUMBR)
140       'Set output parameter from routine 510
150 SUM = S5
155 '<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
160 PRINT SUM
170 GOTO 999
500 ' Routine SUMIT
501 ' Input parameters:  NUM5, the number of values to be summed
502 ' Output parameters: S5, the sum of all the values
503 ' This subroutine adds a list of numbers entered by the user.
504 ' The parameter NUMBR tells how many numbers will be entered.
507 ' SUM gives the sum of the numbers.
508 '
510 S5 = 0
520 FOR I5 = 1 TO NUM5
530   INPUT VAL5
540   S5 = S5 + VAL5
550 NEXT I5
599 RETURN
999 END
```

There should be only one RETURN statement in the subroutine. If it is necessary to return from some other line in the code, simply GOTO the line number containing the RETURN for that subroutine. This preserves our "one out" philosophy for modules. This may seem like splitting hairs, but a subroutine with more than one physical RETURN point can be extremely treacherous to debug.

Finally, line numbers ending in 00 through 09 with the same number as the subroutine header can be used for comments about the subroutine. I suggest that you give each subroutine a name, even though the name will only be used within a comment and is therefore meaningless in the execution of the program. However, I find names a lot easier to remember than line numbers. In addition, the name of the subroutine can also be referenced in a comment attached to the calling GOSUB, again helping to identify the actual function of the subroutine. Note lines 110 and 500 in Figure 3.4-8.

Local Variables

One of the general philosophies about variables is that a variable name should never be used for more than one thing. For instance, if the variable I is used to hold the total of a calculation one place in the program, it is poor practice to later reuse I as, say, an index variable for a loop. This is true even if you know that you won't be needing the value of the total again, making the variable name I "free."

An extension of this idea is to create "local" variable names for subroutines. This can be done by appending some type of tag, unique to the subroutine that the variable is used in, to the end of all variables in the subroutine. One simple approach is to number each subroutine, and use the number of the subroutine as the tag. For instance, if you want to use the variable I in the first subroutine that you create, name the variable I1. Then, when you want to use I in the second subroutine, use I2, etc. The only problem with this scheme is that it can be difficult to remember which subroutine has what number.

Another approach is to use a part of the beginning line number of the subroutine as the tag, for instance, all the numbers to the left of the tens place. For a subroutine that begins at line 610, I would use local variables ending in the tag 6, such as I6 or TOTAL6. For a subroutine that begins at line 1310, I'd use 13 as the tag, etc. Notice that the variables for the subroutine in Figure 3.4-8 all end in 5.

In this way, all variables used within a subroutine are guaranteed unique to that subroutine. In addition, a variable can easily be identified with its subroutine during debugging. Simple naming conventions are always helpful for keeping track of things in a program.

Parameters

BASIC does not allow us to define formal parameters in a subroutine. However, input and output parameters can easily be simulated.

For input parameters, simply identify all those variables to be used by the subroutine that get their values elsewhere. Assign them to local variables immediately before entering the subroutine, before the GOSUB call to that routine. From that point on, never refer to the variables that act as actual parameters, but only to the local variables that the actual parameters initialized. In the example subroutine, *NUMBR* was the actual parameter, while the local variable *NUM5* is used as a formal parameter. Notice that *NUMBR* is never referred to again within the subroutine. *NUM5* is used instead.

For output parameters, simply assign them the values of local variables immediately after returning from the subroutine. Never refer to the actual parameters in any other part of the subroutine code; use local variables instead. In the example, *S5* is used to make the calculation of the sum, but the final value of *S5* is assigned to the actual output parameter, *SUM*, immediately after the GOSUB call to that routine.

If a particular variable is to be both an input parameter and an output parameter, simply do both the input initialization and the output setting.

Finally, as a reminder, identify all input and output parameters in the comments at the beginning of the subroutine. Also, I find it helpful to include a list of the parameters in the comment attached to the calling GOSUB statement (see Figure 3.4-8, lines 130, 501, and 502).

3·5 OPTIMIZATION

Up to now, we have been concerned with the maintainability of a program and have introduced many techniques to improve that quality. This has not come without cost, however. In general, the gains in maintainability have been made at the expense of the efficiency of the program.

There are two types of efficiency to be concerned about. The first is speed, i.e., how fast the system is able to execute the code of the program. The longer a program is, and the more complex it is, the more time it will take the computer to execute the required instructions.

The second type is space efficiency. In this case, we are more concerned with how much memory a program requires. If the program you are developing is approaching the upper limit of the memory size of your computer, it will somehow have to be condensed.

Unfortunately, not only do we trade off efficiency when we use structured techniques, but we also usually have to trade speed for size, and vice versa. Balancing the two requirements can, at times, be very tricky. Usually, however, the choice of which to take, speed or size, is obvious. In the rare occasions where both speed and size are a concern, you should use assembly language for developing your program.

Speed Optimization

Because of the nature of interpreted languages, BASIC is easier to optimize in some ways than Pascal. This is because every line of code executed in a BASIC program must be translated into machine language each time the line is encountered. The code being executed is the source code, i.e., the BASIC instructions themselves. In a compiled language such as Pascal, the entire program is translated into machine language at once, and it is this machine language version that is then executed. Such a compilation performs some automatic optimization of the code. For instance, all comments are removed during compilation. This gets them safely out of the way.

For BASIC, then, there are two easy steps that can be taken that may speed up execution significantly. The first is to remove all comments from the code. This speeds up the program because otherwise the interpreter would have to explicitly ignore each comment every time it was encountered during the execution.

It is not necessary to manually remove all of the comments you have worked so hard to add. Indeed, it is important to keep a version of each program with all of its comments intact for maintenance purposes. The program given in the appendix will remove all comments from a program. Use this compression program on a *copy* of your program to create the version that will actually be executed.

The second method of speed optimization for BASIC programs is to move all subroutines and user-defined functions to the beginning of the program. The best spot is probably just after the program header. This speeds up the program because most versions of BASIC must scan the entire program looking for the line number of any subroutine you call with a GOSUB. Putting the subroutines at the beginning means that they will be found faster.

I didn't recommend putting subroutines in this location when we discussed the program format because I feel that it makes the program a little more difficult to read. In addition, it requires placing a GOTO statement prior to the first subroutine, which will branch to the first statement of the instructions section of the main program. This is quite unappealing, from a structured point of view.

For both BASIC and Pascal, there are two more possibilities for increasing the execution efficiency of a program. The first is to remove as many subroutine calls as possible, moving the code directly into the main program instead. This is certainly obvious for modules developed as subroutines that will be called only once in the program. For subroutines that are called from more than one location in the program, copying all of the subroutine code into each place where it is needed would be tedious. However, it will definitely increase the speed by eliminating the overhead involved in making a subroutine call.

Finally, eliminate as many references to external programs and routines as possible. In the case of BASIC, this means eliminating chaining and merging, and including other code files. This means merging the source files of programs, not waiting until the program's execution for this. For Pascal, eliminate calling other programs that have been compiled separately.

Space Optimization

Again, attempting significant optimization of the space requirements of a program can be extremely complex. Luckily, there are a few methods that can be applied that will provide some relief:

1. Comments should be eliminated from an interpreted language program in order to conserve space. Each character in a comment takes up 1 byte of memory. Comments can easily increase the size of a BASIC program, for instance, by as much as fifty percent. Again, removing comments from a program that is going to be compiled will not have any effect on the amount of memory it requires.
2. Minimize the size of all data structures as much as possible. Don't

make an array's size larger than it really needs to be. Eliminate extra arrays if possible.

3. Use files instead of internal data structures. If it is possible to keep data in a file for use, don't use an array to store it in memory. Read a file only one or two records at a time. Don't read the entire contents of a file into memory all at one time unless absolutely necessary.
4. Although it violates previously discussed guidelines, when necessary you should eliminate duplicate code by using subroutines.
5. Using global variables can save some space. This should be handled with extreme caution, however. The most likely variables to make global are complex data structures such as arrays and records.
6. The net space that a program requires can be regulated using what are usually called **overlays.** This means breaking the program up into segments. Only one segment is kept in memory at any one time. When the code of another segment is required, it is read into memory, overlaying the memory used by the previous segment. The code of the previous memory is therefore no longer in memory. Such overlays can be created in BASIC using CHAIN commands. This method is greatly facilitated through the use of modularity, since segments can be easily constructed out of modules.

4 Human-Engineered Programming

4·1 INTRODUCTION

What are the characteristics of human beings that must be considered when developing a user interface? We must somehow take into account the various psychological factors that affect a human's understanding of a given situation. It is always dangerous to use an "average" psychological profile of a user, since the deviations from this average can be quite wide. However, it is still necessary and possible to identify certain factors that are of consistent concern when dealing with users.

These factors can be easily identified by simply observing users at work. The techniques that a system uses to communicate to the user, and the user employs to communicate to the system, can be observed in action to reveal what users find annoying or helpful. Experimenting with several types of interfaces with different types of users should then make it possible to isolate the good and the bad techniques.

This has, in fact, been done informally by software designers for many years. As a result of these trial-and-error experiments, the attributes of "good" user interfaces have been identified fairly well.

It turns out that many of them are embodiments of simple common sense, the poor man's psychology. The fact that they are not incorporated into more programs can only be attributed to unconcerned managers or lazy programmers.

Operator Skill

After the specific audience of a system has been identified, the user interface should be devised to match the skill level of the expected users. This is certainly one of the most time-consuming techniques for a programmer, especially if it is necessary for the same system to address more than one class of user.

For the novice, it is important to "lead the user by the hand," providing numerous examples. This must be carefully handled so that jargon is minimized. The programmer must assume that the user at this level knows nothing about how to use any system and keep constantly in mind that the user will be confused by every decision that must be made.

To minimize confusion, the programmer needs to anticipate what information the user must have in order to make a decision. For experienced users, the information may be trivial. For the novice, however, it will usually be substantial.

It will often be necessary to create a dynamic system, one that changes as the user becomes more experienced with it. This often requires breaking the system up into multiple programs, one for each level of user. A feature that provides gory details for a novice will quickly bore a more experienced user.

User Reinforcement

Although the normal path for users is to progress from novice to middle-class user, that transition may not be discrete. It is not reasonable to assume that one day the user is a novice, and the next he or she is experienced. In addition, even an experienced user might want to use a feature he or she has not tried before, thereby regressing to the novice stage briefly.

This should lead to a system that, at the middle-class level, allows the user to enter a novice state for a short time, and then reenter the middle-class state. This transition should be simple and obvious. It is not fair to make the user be in either the novice class or the middle class exclusively.

In addition, it is important that the user always know exactly where he or she is within the system. There should be obvious signposts to point out the possible paths to be taken at any time, for more experienced users as well as novices. As we will discuss in a later section, the exact style of the signposts may be different for each class of user, but something should exist for every level.

A common technique for providing this facility is the **menu.** In this type of interface, the user is presented with a list of functions that can be performed. The user then selects an item from the menu. Menus are not difficult to develop, and can offer a great deal of reinforcement to the user. In addition, it is a simple way of letting the user know exactly where he or she is in the system at any time. Most systems that are described as "user friendly" are menu-driven.

We will not discuss the specific techniques for creating a menu-driven system, since such techniques are very computer- and language-dependent. The reader is instead referred to the sources listed at the end of this book for additional reading.

Feedback

Another way of providing the signposts needed by a user is to provide constant feedback about what the system is doing. This is especially true if the system is currently involved in a procedure that will take a significant amount of real time to accomplish. The user should always be informed when the system will be busy for a while. A message on the screen lets the user know that the system didn't just go off into limho.

In addition, it is important to keep the user informed about every action that has been requested, no matter how simple the acknowledgment. Such feedback lets the user know what the system is doing, giving him faith that what he has requested has indeed been accomplished. Given the many suspicions that novice users tend to have about computers, this is quite important.

Finally, all feedback should provide only positive reinforcement for the user. Derogatory messages and flippant remarks may seem like fun to the programmer but will definitely not be appreciated by users. When the user has done something incorrectly, identify as nearly as possible *exactly* what has been done wrong. In addition, suggest to the user how to correct his or her mistake. This keeps the user from having to guess what to do next.

Consistency

Ambiguity in a program reduces the effectiveness of the user, requiring him either to make a decision with incomplete information, or to waste time searching the documentation for a resolution to the ambiguity. Therefore, rules should be established for interacting with the user, and these rules should be followed religiously. This eventually provides the user with the means to make assumptions about how things work in the system.

A negative, albeit trivial, example is the local automatic bank teller machine. Some of the messages it displays are surrounded by quotation marks, while others are not. I have not been able to discern any difference between these quoted and unquoted messages. This is only slightly annoying to the purist in me, but I can certainly understand how someone less familiar with computers might be confused about this difference in message formats.

Consistency will be discussed in more detail in the Designing Outputs section later in this chapter.

Memory Demands

Finally, it is important to minimize the demands on the human memory as much as possible. It has been shown that humans have a remarkable capacity for remembering the most trivial details. However, human memory is also extremely fickle. We don't seem to have much control over which details we remember.

The system should not assume the user remembers every instruction he or she has been previously given. Using abbreviations is, therefore, to be discouraged. The problem is that the same abbreviation used in two different systems would probably have entirely different meanings. This would prove confusing to any user, no matter how experienced.

A mechanism for providing some type of **help facility** for the user is the ideal way to limit the memory requirements. Such a facility can usually be entered at any time by the user. It provides details about how each part of the system works, what the various commands of the system are, and what the formats for inputs are. Help facilities can be very complex, and they are usually reserved for large systems such as word processors or spread sheet packages. The reader is again referred to the sources at the end of the book.

The least that should be provided, as noted earlier, is some mechanism for the user to return to a lower level. Returning to the novice level, for instance, when faced with a confusing situation should make it possible for the user to get more details about what he or she is expected to do. However, it is necessary that such a mechanism be quite smooth and easily used. The user should be able to return to his or her original level without difficulty.

This technique is often handled by setting up a special key or command that the user enters to get help. The input routines used throughout the program must then **trap** (look) for this key or command, sending the user into a sub-module that provides the help facility. The most fundamental system would simply output a long list of instructions for the user in the hopes that the user's specific problem has been anticipated and included in this list. The most sophisticated system would allow the user to request additional information on a particular item. In this way the user would not have to be bothered with the details of unrelated items. Leaving the help facility should then return the user to the exact spot he or she left to enter the help sub-module.

4·2 DEALING WITH ERRORS

There are two main concerns when developing software using human-engineering principles. The first is that the user interface is as natural and comfortable to use as possible. The second, which is just as important but, unfortunately, all too often forgotten by software developers, is the **error handling** capabilities of the system.

Robustness

The objective is to develop a system that thoroughly handles most, if not all, errors that might arise during the use of the system. Such a system is called **robust.** The system should never have an error that leaves the user hanging, not knowing what to do next, but should trap such errors and let the user know, in as much detail as possible, what went wrong.

One characteristic of a robust system is that the program never lets the operating system generate a message. For instance, in BASIC programs, a frequently occurring message generated when an input error is made is "REDO FROM START." This cryptic message is extremely difficult for novice users to understand and usually leaves them in a state of total confusion. A robust program would prevent this message from ever being generated by trapping the input sufficiently.

Another characteristic of a robust system is that it helps the user identify and correct his mistakes. Identifying errors is often difficult because it requires locating the source of the error, rather than where the error manifests itself. (This is discussed in more detail in Chapter 5.) In addition, the robust system should offer suggestions to help the user correct his or her mistake. None of this is easy, and such features can cause even a simple system to become quite large and complex. The programmer must, therefore, make a conscious decision about how much support the system will give to the user, i.e., what level of robustness the system will maintain.

Types of Errors

There are several ways that errors can be classified. We will look at two such classifications.

To begin with, we can identify errors as either fatal or non-fatal. **Fatal** errors make it impossible for the user to continue. Such an occurrence is often called a **system crash.** When this happens, usually all work done with the system up to the point of the crash is lost. This is obviously the worst possible event.

When a **non-fatal** error occurs, it is possible for either the system or the user to correct the error and continue. A non-fatal error is usually the result of some mistake the user has made, such as entering the wrong date.

A second way to classify errors is according to cause. There are three main causes of error. The first is a bug in the program itself, i.e., **an error in the code.** This type of error is introduced by the programmer and cannot usually be dealt with by the user. This is certainly the most frustrating type of error for the user, because the system usually completely crashes or gives incomprehensible results.

Programmers must necessarily promote the notion that programs are infallible. This gives users the faith to use computers. Otherwise, users are too quick to suspect a bug, and do not spend enough time looking for errors they themselves have made. However, such blind faith makes a program bug much more devastating for the user when it does occur. Having one's deepest faiths shaken is never a pleasant experience.

This is the main reason that all the other techniques discussed throughout this book are so important. They are the most active means currently available for developing software with as few bugs as possible. They are aimed directly at preventing programming errors, and so directly benefit the user.

Next, errors can be caused when **entering data.** These are obviously user errors. User errors should always be non-fatal. Unfortunately, few systems successfully trap (i.e., find and handle) all types of errors that a user might make. Any user error that is not found can cause a fatal error, which leads ultimately to user dissatisfaction.

Such data errors can occur in two ways. The first is that the user has made a **physical mistake,** such as entering the wrong date or not entering the date in the proper format. This is the simplest type of error to trap and will be discussed in detail in the Designing Inputs section later in this chapter.

Much more difficult to identify are **logical mistakes.** In this case, the user has been asked for a particular type of data that must follow specific rules. Consider, for example, a program that will classify types of triangles, such as right (having one ninety-degree angle), scalene (no sides of equal length), isosceles (two sides of equal length), or equilateral (three sides of equal length). The user is supposed to enter three Cartesian coordinate points, and the program would then identify the type of triangle created by those three points. What would happen if the user entered three points that make up a straight line? There would be no triangle that could be constructed by these points. Thus, a logical inconsistency results.

Another example of a logical error is trying to use nonexistent or empty files. Such a situation could result from a physical error, such as entering the wrong file name because of a typing mistake, but it could also be the result of an error that erased a file or some anomaly that caused it to be empty.

Finally, errors can be generated by **calculations.** This might take the form of non-fatal errors, such as in the case of erroneous output resulting from truncation or rounding errors. Fatal errors could also result, however. An example is a "divide by zero" error, where the divisor of a division operation is zero. This particular example is always fatal.

Calculation errors are somewhat more difficult to deal with than the other types because they are often dependent on the type of computer and programming language being used, as well as on the specific subject of the application. These factors can affect the precision of variables, which is often responsible for calculation errors.

How to Deal with Errors

Now that we have some idea of types of errors, we can move on to the various concerns for handling errors. Three main areas that must be addressed are: preventing errors, detecting errors, and correcting errors.

Preventing Errors

Most of what we have been concerned with up to now has been preventing programming errors in a system. The techniques discussed in Chapter 5, Program Testing and Debugging, further support this effort. In addition to preventing programming errors, however, we must engage in **defensive programming.**

This technique requires the programmer to assume that the user of the system *will* do something he or she isn't supposed to do. If, for instance, you indicate that a particular input cannot be negative, you must be prepared for the user who will, in fact, enter a negative number. This technique greatly complicates input routines, but it is essential to eliminating the largest source of errors, bad user input.

The essence of this approach is to anticipate the mistakes that users will make and to develop screening routines to filter out bad input. We can identify input as "bad" by relying on the data definitions of items for which we have set up variables. Recall from section I.4, Dealing with Data, that data can be defined with a variety of attributes, the main ones being value, type, range, and precision. The ones of concern for developing effective input screens are type, range, and precision. With the data definitions providing details about these attributes for each data item used within the system, we can easily establish whether or not a particular input conforms to its defined attributes. If not, then the user has obviously made a mistake.

Detecting Errors

Discovering that an error has occurred in the code is much more difficult than eliminating the most obvious input errors by using screens. The problem is that an error could be present at any time during the execution of the system. Locating the source of the error is difficult even for human beings to accomplish efficiently. Since computers are not (yet) self-aware, it is extremely difficult for a program to identify errors in itself.

We cannot hope to detect all types of errors that develop despite our best efforts to prevent them. We can, at best, look for certain kinds of logical and calculation errors.

Logical errors will be highly dependent upon the purpose of the system. For instance, in the triangle program example given above, the mechanisms devised for detecting logical errors will be quite specific to geometry. It is necessary, therefore, that the programmer have a firm understanding of the program's subject matter. This is not always easy, but will go a long way toward creating a system that can detect logical errors.

Other kinds of logical errors are easier to trap. The most obvious ones have to do with logically empty conditions. This occurs with files when a file does not exist or is empty. It also occurs with loops when, for instance, the initial value of the loop control variable already exceeds the final value given in the loop header statement. This may be perfectly all right in some contexts, but it is deadly in others. Loops should be scrupulously examined for such problems.

In the case of calculation errors, the programmer must again anticipate what type of errors might occur. For these errors, we are still concerned with the data and its attributes. Key data should be checked at strategic spots throughout the program to verify that the values of that data fit the attributes defined. "Strategic spots" would include after complex calculations, upon entering or exiting modules, or prior to outputting the data.

The only other calculation type error that can easily be detected is dividing by zero. It is not too difficult to add code along the lines of:

```
IF divisor is zero THEN call error routine
      ELSE result = dividend / divisor
```

This is the only sure method of eliminating this type of error.

Correcting Errors

This is the most difficult area of dealing with errors. Even if the program is intelligent enough to detect a wide range of errors, it is seldom possible for the program itself to correct errors.

For errors that have been created by the program itself, there is little that can be done. In the case of a code error, it is impossible for the computer to change itself so that the error no longer exists.

The programmer must obviously be involved in this level of correction.

It might be possible for a program to correct some types of calculation errors. For instance, errors in type or precision might be correctable if the program knows how to translate from the present form of the data into the correct form. This **massaging** of the data is not easily accomplished, however.

For errors in value, including range, there is usually not much that can be done. While it is possible to force a data value into a specified range if that value has suddenly gone outside of that range, what should the new value be? There is no heuristic that the programmer can use to determine this in all cases.

Most errors that are correctable have been made either by the user directly, as in the case of entering bad data, or indirectly, through calculations made with the data supplied by the user or obtained from files. For this reason, most methods that attempt to correct an error in the data involve the user in some way.

The simplest approach, and the one taken by the vast majority of programmers, is to output some type of error message and end the program. This raises the classification of user error involved from non-fatal to fatal. While this may be necessary for errors that have occurred in data files or were created by calculation, obviously this is not appropriate for errors made by users entering bad data. The program should, instead, enlist the aid of the user in identifying the source of the error and correcting it. This is obviously a difficult procedure when dealing with data file and calculation errors, since the source of the error may be far removed from the knowledge or expertise of the user. This approach is usually only possible when dealing with highly sophisticated users, in the upper-middle or expert class.

In the case of calculation errors, however, it may be possible to trace them to data that the user entered. At the very least, a program can require the user to verify all of the data he or she has entered up to that point. The program can, perhaps, run additional diagnostics on that data to ensure that it is not the source of the current problem. If the error can be traced to user data, the correction procedure is obvious.

As mentioned previously, it is usually possible to construct some type of screening mechanism for keeping bad data out of the system. The procedure is quite simple, and involves a loop that allows the user to correct any mistake that is immediately identified by the system. Such loops have already been used in the algorithm examples in Chapter 2. The general format of such loops is:

```
REPEAT
        input a value from the user
UNTIL the value is valid
```

This procedure requires that at least some of the attributes of the data are known and can be tested for. A simpler form of this is sometimes called a **feedback loop,** because it feeds the value entered back to the user for visual verification:

```
REPEAT
        input value from the user;
        output the value to the user
UNTIL user verifies the value.
```

This form of the error trap is useful especially when the attributes of the data being input are too broadly defined to be of much help. In addition, this is a very important technique for eliminating typing mistakes by the user, a very large source of errors. The method is not always possible, however, since the way data is entered can affect the ability of the program to trap the input. For instance, it is neither desirable nor possible to verify the input of a joystick in most applications. The error trap method described is appropriate for any input that can be essentially considered as discrete. This would include any input from a keyboard, most input from a mouse, and only some types of input from a joystick.

The best approach is to combine the two loops above into a single trap:

```
REPEAT
        input a value from the user;
        output the value to the user
UNTIL (the value is valid) AND (user verifies value).
```

This covers all bases. Note that the program's validation of the data occurs before the verification by the user. This is appropriate, since there is no need to bother with the user verification if the data entered does not conform to the proper format.

Some type of error message should be generated when an error has been detected in the system, even if the program itself is going to make an attempt to correct it. The art of creating understandable messages will be discussed in section 4.4, Designing Outputs.

4.3 DESIGNING INPUTS

This section will expand upon this concept of preventing any data that the user enters from causing errors in the system. In addition, we will be concerned with creating one-half of the user interface for a system, namely how the human user communicates with the machine. In this case, we will want to take data in a form that is understandable to a human and translate it into a form the computer understands. We will not be so much interested in this translation process as in what the user understands he or she is to enter. This overlaps somewhat with the design of appropriate outputs, since we need outputs to tell the user what to enter.

Communicating with Machines

We would like to be able to simulate the various modes that humans use when communicating with each other—sound, sight and touch—so that humans can use those natural means when communicating with a machine. The ideal is to create a user interface that a human could use in exactly the same way that he or she communicates with other humans. Unfortunately, this requires not only advances in the current technology used for communication, but also advances in the way a computer understands things. The area of artificial intelligence is deeply involved in this type of research. Until this ideal level of communication is possible, it will be necessary to construct devices that simulate, as well as possible, these natural modes of human communication.

Characteristics of Communications Devices

Let's first examine some of the characteristics of the various devices that can be used by humans to enter information into a machine.

First, we can classify devices as being either direct or indirect. Each time a **direct** device is used, there is an immediate, observable effect on what the program is doing. The user associates the device being used directly with the action that occurs.

An **indirect** device is more passive in nature. The device itself is not readily identified with a specific function. As a result, such a device is largely symbolic. The circumstances surrounding the meaning and use of such a device are largely context-dependent. As a result, the meaning associated with an indirect device changes not only from application to application, but within a single application as well.

Next, a device is either discrete or continuous in nature. In a **discrete** device, the values entered are exact and individually distinct, and the user must recognize the discrete values that may be entered. A watch with a digital display gives time the appearance of being discrete, in that it projects an exact time, usually down to the second. Most inputs will use discrete values, since this is easier for humans to understand.

A **continuous** device enters values in a theoretically infinite range. Real numbers in mathematics are continuous in nature, since you can have an infinite number of decimal places. Floating point numbers in a computer are really discrete, since there is a limit to the number of decimal places stored. An analog watch (one with hands) shows time as a continuous quantity, which it truly is. This situation is similar to the real number concept, in that time can be divided as minutely as we wish, even though doing so may be of no practical value.

Most continuous inputs are approximated with discrete values in a computer system. However, we can consider some devices continuous because they give the appearance of continuous inputs. This illusion is seldom a problem. Truly continuous input devices do exist but are not well-suited to human use.

Finally, we can consider some inputs to be **real-time** inputs. In this case, the inputs have an immediate affect on the execution of the program. The term *real time* refers to the fact that the computer is expected to respond to the input within the objective time of the user. A real-time system must be able to react to its environment quickly enough to affect its own inputs.

Where there is a human user involved, this real time is dependent upon the user's reaction to outputs from the system. Therefore, the user must be able to enter information into the system as quickly as possible. There is an implied feedback loop containing the user's visual system and the method of entering data. Since most input devices require using the hands, this generally implies a high degree of hand-eye coordination. The system must react within a period of time that appears normal to the user. Events displayed on the screen must not be so fast that the human eye cannot record them, the human brain interpret them, or the human nervous system react to them.

As a result of the unreliability of human response time, most real-time systems do not include human users in the feedback loop, but are instead hooked to sophisticated sensing devices that provide the necessary inputs. This is usually called **process control.** An example of this would be an automatic pilot for an airplane. However, there are examples of real-time systems that do include human users directly. The most notable are video games.

Audio Devices

In the case of the audio mode, there is, unfortunately, little that can be done. While there are some exciting prospects in the field of speech recognition, most developments are still experimental. This is because, even though we might all use the same words, other attributes of speech make it extremely difficult for a computer to recognize sounds as words. For instance, speed, voice inflection, and accent can all change the characteristics of the sound produced by the human vocal system. Unfortunately, the computer's inability to interact with the human voice eliminates the most natural form of communication for human beings.

Other types of audio communication are not appropriate for developing the input portion of the user interface. Although with training a human can produce beeps and whistles easily recognizable by a computer, such a method of input would hardly be natural (at least for most people).

For the time being, these problems completely eliminate audio input for communicating with computers.

Visual Devices

Devices that rely upon the visual ability of a human are typically text-oriented. They most often require the user to interact with the system using words, either in a natural language or, more likely, using an artificial vocabulary.

However, some of the more interesting systems use special symbols called **icons** to communicate with the user. These icons graphically represent actions that the machine is able to perform. Instead of using words to instruct the computer to perform these actions, the user merely points, in some manner, to the proper icon. This informs the computer that the user wishes the function designated to be performed.

TEXT DEVICES It is possible to simulate at least one method of visual communication for use in the user interface. Since written languages have a definable vocabulary, all that is required is to create a device that simulates the human capability of writing by hand. A typewriter-style keyboard can obviously be used for such a purpose.

The keyboard is the most frequently used and most misunderstood input device. There are several reasons for this. First, its flexibility makes it useful for a wide range of applications. Unfortunately, it is this flexibility that is the cause of its misuse. It is an inherently difficult device to master, requiring great manual dexterity to use efficiently.

Second, there is no standard design for a keyboard. Although the arrangements of keys in the QWERTY pattern is standard, the size of the keys and their spacing is not standardized. In addition, computer keyboards have many specialized keys not found on typewriters. The arrangement of these special keys is at the complete discretion of the manufacturer. To make matters worse, the set of special keys is not even a defined standard. Few manufacturers' keyboards have the exact same keys.

Third, the names and the functions of these special keys are often quite mysterious. This is especially true for keys that seem never to be used. Names such as CTRL, ESC, and PF1 do not lead to ready identification of a particular function for those keys. Even more confusing is that the function of certain keys can change from one application to another.

Fourth, the keyboard is mostly an indirect device, in that nothing happens until all the keys are properly struck. The keys must usually be combined in order to impart any information. It is very difficult to attach any particular meaning to a keyboard. All work with the keyboard is essentially symbolic. Nevertheless, the special keys *can* sometimes take on the characteristic of being direct devices, thereby adding to the confusion.

Fifth, the keyboard is obviously a non-real-time device. Usually nothing else can be going on in the system while the program is waiting for input. This is fine for most applications. However, when real-time input is required, some other device will need to be used.

Finally, it is too easy to make a mistake using a keyboard. It is difficult to learn where all the keys are so that they can be struck proficiently. Changing keyboards can require relearning the location of the keys as well as a readjustment of the movements necessary to strike them accurately.

Because the keyboard is such a flexible device, special forms of the keyboard can be designed for specific functions. The most commonly used alternate keyboard is the numeric keypad, used to enter large quantities of numbers. Other keyboards can be designed for other special purposes.

There is even a keyboard that rearranges the alphabetic keys so that the most frequently used keys are located directly underneath the fingers. This is called the Dvorak keyboard, named after its inventor. He developed this style of keyboard because the most commonly used format, the QWERTY keyboard, was designed specifically to slow typists down. Because of the technology that was available when the typewriter was first implemented, a typist who was too fast would cause the keys to lock up. Spreading out the keys in the QWERTY format slowed most typists down enough to avoid this misfortune.

POINTING DEVICES A mouse is another type of visual device. It is becoming increasingly popular because it allows some inputs to be done in a (seemingly) more natural way than a keyboard. Although it is technically a continuous device, it is most often used in a discrete mode to allow the user to point to various discrete boxes or icons being displayed on the video screen. The mouse is used to position the cursor on the icon. The user must then push a button on the mouse to direct the program to perform the function indicated by the icon.

Using icons and a pointing method is a very direct way of communicating with a computer. It provides a general device, a pointer, and a special device, the icons, developed specifically for a particular application. Using the mouse pointer is very similar to using special function keys on a keyboard. The beauty of icons, however, is that they graphically represent their functions so that the user can easily learn to associate a function with its icon. This is certainly at the heart of the mouse's growing popularity.

There may be one or more buttons on a mouse. Unlike the special keys on a keyboard, these buttons often have specific meanings that are independent of the application being used. Unfortunately, some manufacturers' concepts of a mouse are different from others, mostly in terms of how many buttons a mouse should have. Since the mouse is used by one hand, some designers have placed as many as five buttons on their mouse. The most commonly used design appears to be the three-button mouse. At least one computer company has decided that only one button is needed. Unfortunately for them, they have discovered that, as a result, some operations require the user to press keys on the computer's keyboard in addition to the button on the mouse. This makes what should have been a one-handed operation a two-handed operation, a definite disadvantage.

In addition, because the user can observe an immediate response from the system when the mouse is used (i.e., the cursor moves "immediately" when the mouse is moved), this is sometimes thought of as a real-time device. However, since the *user* does not need to react in real time, even though the computer does, the mouse is not usually considered a true real-time device. Nothing happens in the application until the user presses a button on the mouse to inform the computer that the current position of the cursor points to the function the user wants performed.

There are two other devices that can be used in a pointing mechanism. The first is a rather old device, the joystick, which was used as a pointer long before the mouse became fashionable. It has many of the same characteristics as the mouse when used in this manner. Most joysticks include at least one button. The main reason the joystick is not used instead of the mouse is that the physical characteristics of the joystick do not make it as easy to use for pointing. It must either be firmly mounted on some solid surface or held in two hands. In addition, on many joysticks, two hands are required to use both the joystick and the buttons at the same time.

However, many forms of the joystick can be used in exactly the same way as a mouse to point to icons on a screen. In addition, the joystick can be used as a real-time input device. This is obvious from game programs that use the joystick to manipulate figures on the screen immediately when the stick is moved. This makes the joystick especially suited for use in simulation programs, which are typically real-time in nature.

Another technology that is enjoying a rebirth for use as a pointer is the touch screen. With this device, the user simply points at the icons on the screen with a finger. This makes the device mostly transparent to the user, since the sensing device is built into the screen and no obvious device or skill needs to be mastered.

This mechanism enjoys most of the same benefits as the other pointing devices described above. However, its biggest disadvantage is its lack of resolution and, therefore, precision.

The touch screen employs a method of dividing the screen up into predefined blocks. Any of these blocks can be referenced individually within an application. The system can detect the finger within any of the blocks. Unfortunately, the blocks must be large enough to accommodate the typical human finger. In addition, the blocks must be large enough that the human user knows precisely which block he or she is currently pointing to, without accidentally allowing his or her finger to enter one of the surrounding blocks. As a result, a reasonable block size might be one inch square.

A typical video screen might be about ten inches by seven inches, allowing for, at most, seventy distinct points that can be "touched" on the screen. This sounds like a lot until you realize that most icons cannot be easily represented in a one square inch box. It often requires several boxes to represent the icons used. In addition, it is necessary to space the icons sufficiently so that the user does not easily enter the defined space of the wrong icon accidentally.

Finally, another device that can be used easily for pointing is the light pen. With this device, the user points to the screen with the pen. The computer is able to detect where the screen is being pointed to using light-sensitive diodes and a coordinate system very similar to the touch screen mechanism. It is a much more precise device than the touch screen, however, since the area pointed to by the light pen can be much smaller than that necessary for the touch screen. Used in this fashion, a light pen is a discrete, direct, non-real-time device.

GRAPHICS DEVICES The joystick and the light pen can both be used for entering graphical data into a system. For such use, the devices are used in a continuous, rather than a discrete, mode. This is because lines must be considered continuous, even though we know that, when represented using any graphical device, they are actually made up of individual dots. Our eyes perceive these dots as a solid line whenever the dots are close enough together.

Another device, sometimes called a digitizer pad or a graphics tablet, also provides a means to enter graphical data. In this case, a touch-sensitive pad is used with a special stylus. Moving the stylus across the pad enters the points crossed on the fine grid of the pad. This grid is similar to that used for devices such as the touch screen. The resolution of this device is higher than can be achieved with most other devices, however, and is much more easily controlled.

This, too, is a direct, usually continuous, non-real-time device. The qualification "usually" here is because the types of devices being described as continuous to this point are really only simulating continuity using discrete technology.

Tactile Devices

A truly tactile device would be one that could sense a change in the amount of pressure exerted on it. This would require a continuous device. However, since we can simulate continuous devices with a number of discrete devices, we can also simulate tactility.

The most notable example of a tactile device is the joystick. In a real-time mode, the distance that the stick is pushed in a particular direction serves as an indication of the degree of change that should take place in the program. For instance, in a simulation of an airplane, pushing the stick forward can be used to force the nose of the plane down. The further the stick is pushed forward, the faster the nose goes down.

Using the joystick in this fashion requires getting the proper "feel" for how far the stick must be pushed in which direction in order to achieve the desired effect. This is the first important factor to be mastered in learning to play a video game, and will be important in any simulation that uses such a real-time device.

Other devices can also have this tactile quality without being considered real-time devices. For instance, a graphics tablet can be used to draw pictures. The pressure used when moving the stylus across the tablet can indicate how sharp a line is drawn, or how deep a color is "painted." The user must learn to use just the right amount of pressure in order to create the desired picture.

A light pen, to a lesser extent, can be used in the same way. However, the lower resolution of a light pen makes it more difficult to use as a tactile device.

Techniques and Guidelines

Now that we have explored some of the characteristics of devices, and examples of specific types of devices, we can at last begin discussing how inputs can be designed in order to support the user as well as possible.

Choosing an Input Device

The first task is to decide which input device will be used for each input that must be made. The objective is to use the most natural device for entering the particular information involved. This is often at odds with expedience, however, since not all computers are equipped with a wide range of input devices. When alternatives are available, the programmer should not choose the keyboard as the sole entry device simply because it is the easiest to program for.

Since the vast majority of inputs will be done using a keyboard, the rest of the methods discussed here will deal with using this inhospitable device. Using the other devices for input is more a matter of the mechanics involved than the philosophy of the device's use.

Command Entry

There are two main types of input that the user must enter. The first is instructions to the application. Any such instruction will simply be called a **command.**

A command is any indication by the user that he or she wants a particular function performed by the system. The set of commands for any application is contrived entirely by the programmer, who must devise commands that do not confuse the user, are readily understood, and are easily remembered.

First, commands should have simple, yet meaningful names. This directly aids the user by making the commands easy to remember. Names should not be too similar in spelling, sound, or meaning, since this can lead to confusion for the user.

Second, allow abbreviations for the names of commands, especially long ones and those that are used frequently. Abbreviations should be more than a single letter, and should still attempt to be meaningful. A common technique is to use the first few characters of each command as the abbreviation. Obviously, the number of characters used must be sufficient to ensure uniqueness in the abbreviated names.

Third, use special function keys to indicate commands whenever possible. This allows the user to set up templates that describe the function for each special key used. Not only is this method of input quicker for the user than typing in a word or phrase, it also makes it unnecessary for the user to remember the exact command. He can instead concentrate on the function that he or she wants performed.

If the keyboard being used does not have special function keys, do not substitute control codes, however. Holding down the control key and some other alphabetic key is an unnatural action, and is difficult for some combinations, depending on the location of the control key. In addition, there is often no connection between the keys used and the function being performed, so that there is no memory aid for the user. Use whole word and abbreviated commands instead of control codes.

Fourth, display the list of commands for the user whenever possible. This at least reminds the user what the valid commands are. Even better is allowing the user to select from this list, or menu, the command he or she wants to perform.

Finally, allow the user to back out of a potentially harmful command. For instance, for any command that would erase a file, ask the user if he really wants this to happen. This occasionally saves a user from himself when he enters the wrong command or does not fully consider the consequences of an action.

Data Entry

The second type of inputs that a user must enter are various types of data. The principles discussed in this section are independent of the data's type.

INPUT SCREENS To begin with, data entered by a user should be carefully screened to prevent an error in the system. Such screening should verify that the data entered conforms to the type, range, and precision defined for that data item. In addition, the user should be given an opportunity to correct any error.

In BASIC, this requires inputting all data as strings, rather than as a combination of string and numeric data. Entering a string input by accident when the variable on the INPUT statement is numeric causes the system to reject the input and generate the "REDO FROM START" message. As discussed in section 4.3, Dealing with Errors, we don't want the operating system itself to generate error messages for us because they tend to be cryptic and uninformative. Each program should create its own error routines.

This can be a problem in other languages as well. Entering the wrong type of data may cause the program to crash. An equally devastating prospect is that the erroneous input may be accepted, but the value that is assigned to the input variable will be garbage. In such an event, the user may not even be aware that an error has occurred.

FORMAT CONSISTENCY Next, input formats should be consistently handled. If the data to be entered is a date, the format should be the same every place in the program that a date is entered.

Don't require dashes (-) separating the month, day, and year in one place, and slashes (/) in another. In addition, be consistent about when a carriage return is required. Use sparingly inputs that require only one key to be struck. People are used to hitting the return key after each data item, and one-key items do not give the user a chance to change his or her mind about what has been entered.

DEFAULTS Next, it is possible to create **default** values for many inputs. When the user does not enter a specific value, the default value is automatically assigned. The user, of course, must be thoroughly familiar with the defaults of a system in order to ensure that any defaults used are acceptable.

For example, the user might be asked to enter today's date at the beginning of the program. Then, whenever a date is required as input, the user has the option of allowing the system to use that date, which has become the default value, or of entering another date. Another example might be the area code of a phone number. A standard default of the local area code might be used as a convenience if most calls will be local.

There are two main purposes for default values. First, they help the user by being a shorthand way of entering specific values that are used frequently. This speeds up the input process. Second, they help to lower the error rate for data entry, since a default value is always valid data. There will never be a typo in a default value.

However, using defaults in a system can introduce new ways for logical errors to occur. The user may use a default value without realizing it, and that value might be incorrect for what the user is doing. This is especially likely since default values are easy to enter, usually requiring only one or two keystrokes.

As a result, whenever a default value is used, the user should be explicitly informed of the fact. This can usually be handled with a simple warning statement, such as "WARNING: THE DEFAULT VALUE 09/08/84 IS ASSUMED." At the very least, the default value should be displayed and verified by the user in exactly the same way as any other input.

ASSUMPTIONS In every program, the programmer makes some assumptions about the way the program should work, the environment it will operate in, and the users of the system. Many of these assumptions are deliberate, and should be fully described in the user documentation. There is a danger, however, that a programmer will instill many assumptions into a system *unconsciously.* These assumptions never appear in the user documentation, because the programmer was not even aware of them. Nevertheless, the unwitting user can encounter many problems as a result of these hidden assumptions. They introduce a potential source of error and can make a system quite difficult to use.

For example, it is usually assumed—unconsciously because we are used to it—that a person's name is made up of a first name, a middle name, and a last name. However, there are many people who have more than three names. In addition, many men have distinguishing tags on the ends of their names, such as Jr., Sr., or III. Many software systems are only interested in the middle initial, not the entire middle name. But this can cause problems for people without middle names, or those who go by their middle names and use a first initial.

Another source of difficulty arising from assumptions has to do with order of input. We assume that the natural order of entering a person's address is street, city, state, and zip code, and for such a well understood set of information, this order is natural and should obviously be followed by the programmer. However, an appropriate order of input is not always so well understood by the programmer. He must obviously choose *some* order, and will likely choose one that he assumes to be natural. But it is important that the programmer consult potential users to verify that the order chosen is suitable. An order that is perceived by users as awkward will cause a great deal of frustration.

OPTIONAL ENTRIES Finally, values for many data items do not need to be specified by the user when an input is requested. These **optional inputs,** however, can create some dilemmas for the user. For instance, consider a request for a name that specifies that the name should be entered as first name, then middle name, and finally last name. A valid entry would be "Arthur Tyrone Williams." We might even get away with something like "Arthur T. Williams." But what if the person whose name is being entered doesn't have a middle name, or the name is unknown? How should the name be entered? The user was instructed to enter three items, with the middle name as the second item. If the user enters "Arthur Williams," will "Williams" be taken as the middle name?

This can get even more confusing when default values are used in optional entries. The user must be explicitly informed when he or she is allowing the system to use a default value for an optional entry, as well as how to designate that he does not wish to enter a value for that item. For instance, an asterisk (*) or some other special character that is unique within the system could be used as a substitute for an optional value. The method of entering no value for an optional input should be different from the one used to indicate that the system should substitute a default value.

4·4 DESIGNING OUTPUTS

The main object of designing outputs is to supply useful information to the user, provide feedback about errors, and help the user to assimilate the information generated by the system. These functions are not always easily performed, and mean some extra work for the programmer. In some companies, such "niceties" are considered to be expendable bells and whistles. The often-used excuse for not including these features is, "The need for the system is so urgent. A working system without these features is better than a system with these features that is delayed a month."

Such excuses are usually the result of bad planning. The system life cycle of any program should include sufficient time to *fully* implement the system, including all of these "bells and whistles." Good ouput features are an integral part of the system and cannot be added at a later date. Such retrofitting invariably winds up either poorly done or costing ten times more than if the work had been done originally.

In addition, the management that makes such a decision obviously does not understand the saving that is implied by guaranteeing satisfied users. Such users do not inundate the programmers with questions about the system, nor do they make frequent requests for additions to the system. In the long run, a little extra effort at the original implementation phase can save enormous amounts of maintenance work.

Forms of Communication

The computer has three main forms of output. These forms make up the only ways that the computer (and the programmer) has to communicate with the user. Most modern systems, especially microcomputers, can employ all three of these modes.

Text

The standard output device today is the video screen. The printer is a close second. For both of these devices, the main way to display information is as text, where the system generates various types of messages for the user to read. As with the keyboard, the text format for output is the most general form, and is, therefore, the most difficult to use well.

Most programmers use this form of communication exclusively, which would lead one to suspect that programmers have developed text output to a fairly high level. Nothing could be further from the truth. Applying Sturgeon's famous ninety percent rule (Legend has it that Theodore Sturgeon, a well-known science fiction author, was once asked why ninety percent of science fiction literature was so bad, to which he replied, "Because ninety percent of *everything* is crud!") is very easy when dealing with this type of output. It is usually cryptic, ambiguous, often misleading, and almost always riddled with jargon.

Let's now look at the four main uses of text in a system.

INSTRUCTIONS Instructions are a series of narratives that explain to the user how to use the system. Instructions can be either general or specific. General instructions are usually given at the beginning of the program, and explain the general purpose of the system. However, such general instructions can be used throughout the program whenever a new function is entered.

Specific instructions provide the details about what the user is expected to do. These are often developed into an elaborate help facility that the user can consult at any time during the execution of the program. This is an especially effective method of supporting the user.

In order to provide a system that is flexible enough to adjust to the level of the user (from novice to expert), an effective technique is to provide at least three levels of instructions. The first level is used for novices and provides a very comprehensive set of instructions. The second level is intended for more experienced users. It provides an abbreviated form of the instructions. These instructions are somewhat more difficult to write, since they must include the meat of the information in a condensed version yet still be explicit and unambiguous. Finally, the most advanced level probably offers no instructions at all. Unfortunately, this is the level most programmers provide instinctively.

INFORMATION This type of text does not require a response from the user. However, the user is expected to read and comprehend this text, since it is likely that the user's future interaction with the system will be influenced by it. An example of this type of text are headings that describe what the values in a report are.

PROMPT MESSAGES This is probably the most familiar type of text, and is also the one area of output that most affects how a user interacts with the system. This text provides specific messages to the user indicating that input of some kind is expected.

It is extremely important that the user know as much about the input he or she is expected to enter as possible. This is sometimes difficult, given the constraints of the amount of information that can be fit on a screen. The design of prompt messages should be taken quite seriously by the programmer.

ERROR MESSAGES These messages indicate to the user that he or she has either done something incorrectly or caused something to be done incorrectly. Error messages probably confuse users more than any other type of output because most error messages are, at best, cryptic.

Graphics

In recent years, there has been a great deal of advancement in the area of providing more output modes than simple text. The most notable advances have been made in the area of graphics output. These graphics are used to draw pictures that either provide useful information themselves or simply assist the user in understanding what the system is doing.

SHAPES The resolution of the lines that produce shapes is very important in making a picture that is recognizable. The thicker the lines have to be, the lower the resolution of the picture, and the less easy it is to recognize.

According to current, though controversial, medical theories, pictures appeal more to the brain's right hemisphere, while words are interpreted by the left hemisphere. Providing pictures to accompany text, then, gets more of the brain involved. Pictures are also a more direct, immediate method of communicating, since they do not always need to be interpreted. However, even when symbolism is used, the interpretation can be much faster than when words are used.

COLOR Color is used in graphics in a secondary role. Its primary purpose is to help distinguish objects. Color can also be used to help make text more readable, or to make certain portions of the text stand out. For example, error messages might be displayed in red, while the rest of text is displayed in standard white on black.

Finally, color can be used to add character to an otherwise ambiguous shape. This is especially necessary when the resolution of the hardware is low. For example, a tree shape might be more recognizable if it is displayed in green.

ANIMATION Animation can be defined as any visual change in the display. It is most often thought of as being some form of a moving picture, such as is used in making cartoons. This is certainly the most advanced form of animation and is particularly useful in showing dynamic processes in simulation systems that represent real-world events.

However, simpler forms of animation can be used to draw attention to a particular area of the screen. An example of this is a blinking cursor that identifies where on the screen the next output will appear. Some computers and languages allow the user to make words or sentences blink. This can be used, as color is, to make a particular section of text stand out, such as in the case of error messages.

Animation is also helpful in letting the user know that the program is still working when there is a lengthy computation. Using some form of simple animation when there is such a delay informs the user that the program has not simply gone off into limbo. This is often more reassuring than a simple text message.

Audio

The third mode of computer output is audio. The effects that are possible using this mode range from quite simple to very complex. Audio usually plays a secondary role, acting to enhance, reinforce, or "punctuate" the visual output of text or graphics. As in the case of graphics, audio adds to the user's means of interpretation.

NOISE The lowest form of audio output is noise. In a strict sense, noise is an uncoordinated sound that cannot be distinguished by the human ear. However, the main audio output of most computer systems, the beep, can be rightly called a noise even though it is recognizable as a computer beep.

This beep is not associated with any single purpose except to get the user's attention. Its most usual use is to indicate that the user is supposed to do something, such as enter an input. A beep can also be used to warn the user that he or she has done something wrong, such as entering an incorrect input value, or to indicate that the computer is involved in some action. For instance, the system might beep every second or so during a long delay for a complex calculation or while inputting from a tape.

SOUND A sound is any audio output that is recognizable to a user as representing some event, action, or thing. An example would be musical tones. Not every computer system is capable of creating sounds, and even those that can produce sounds such as musical tones cannot usually generate other types of sounds. Sounds add an extra dimension of meaning to other types of output and are seldom used by themselves to convey information, mainly because of the difficulty of conveying factual information with sounds, no matter how elaborate.

The prime example of the use of sound output is the video arcade game. Even those with little experience in playing these games can quickly recognize the various sounds of firing phasors, slamming doors, exploding rockets, or breaking glass. Music is used heavily, and often the music *is* used to convey information, such as the arrival of the villain. Such "lifelike" sound effects can add tremendously to simulations of real-world events.

VOICE The most complex form of sound is voice output. Voice synthesizers have become quite sophisticated and can be used effectively to convey factual information. The more advanced systems are easily understood, even though there are usually a few different syllables that sound alike or are slurred.

Such systems are even quite inexpensive. They can be found in a number of children's educational devices and toys. The famous Texas Instrument's Speak-and-Spell was one of the earliest consumer devices to utilize this technology. Today, even automobiles talk.

However, just because it is possible to synthesize speech doesn't mean that its application will be appreciated by the user. For instance, a local grocery store used such a device to announce the price of each item that was scanned using their new UPC code optical scanner. The synthesizer salesman convinced the store manager that this would make it easier for customers to accept the scanning device. The price of each item was also displayed using the more traditional video display.

It turned out that the customers hated the voice output. First, it was additional "noise" in an environment already filled with noise. Second, the characteristics of the voice itself were annoying. Customers generally found the voice grating. Third, and perhaps the most important point, customers did not like the idea of having their purchases announced in such a way that anyone within fifteen feet of the checkout counter knew what they were buying, or (even worse) how much they were spending. Within a month, the speech synthesizers were removed from the store.

Techniques and Guidelines

Text output is the most commonly used form of communication from the computer to the user. Unfortunately, it is also the most abused as well as the most easily misunderstood by the user, since it cannot help but include all the ambiguities that exist in written language. As a result, we will concentrate on methods that can be used to improve text communication.

Message Outputs

The various messages that a system generates are frequently the source of a user's confusion. Such messages need to be quite explicit and completely unambiguous in order for users to fully understand what they are expected to do next.

PROMPT MESSAGES To begin with, prompt messages should be as explicit as possible about what is expected from the user. Such explanations usually require more than the one or two words that most prompt messages comprise. These cryptic messages usually do more harm than good.

Prompt messages should provide explicit details about the input's format, type, valid range, and necessary precision. In addition, it is usually helpful to provide some example of what the input should be. For example, a prompt message to enter a date might be

Enter Today's Date as Month/Day/Year (e.g. 03/20/84):

An example of an inappropriate prompt message is taken from one of the most popular operating systems. When the user first starts the computer, he or she is faced with a message

```
Current date is Sat 10-01-1983
Enter new date:
```

The first line of this message gives the date that is currently understood by the system as being today's date. The second line is where the user is expected to enter today's date if it differs from the date given in the first line.

There are two things wrong with this prompt. First, it appears that the first line gives an example of the format for the date the user is supposed to enter. A user unfamiliar with this system is likely to enter a new date such as "Fri 09-22-1984." Unfortunately, this will be rejected by the system. The problem is that the user is not supposed to enter the day (i.e., "Fri" in this case), but only the date (09-22-1984). The result is instant confusion and frustration.

The second thing wrong with this prompt is that it does not indicate that the date given in the first line will be used as a default if the user simply hits the return key when asked to enter a new date. Since a novice user undoubtedly does not realize the consequences of allowing the default to be used, this is a potentially dangerous situation.

Presenting complex prompts can often be difficult and can require a lot of text that must be read and understood by the user. This is not always beneficial, since once the user gets used to how particular inputs are handled, he or she will mostly want to ignore the details of the prompts. The only way around this dilemma is to have a multilevel system that can differentiate between novice, experienced, and expert users. This would require constructing at least three different prompts for each input. Even if this is not practical, however, at least two forms should be available, a full description and an abbreviated form.

Since this mechanism usually requires a great deal of space, it is often necessary to remove at least the full descriptive messages from the program. They can be stored on a message file and keyed so that they can be selected randomly. This slows the system down, but, in itself is not a problem for the user who needs the full message.

Next, the prompt messages and their associated inputs should use natural formats, i.e., formats that are the easiest for the user as opposed to formats that are the most convenient for a computer. For example, decimal numbers should be used at all times, even though hexadecimal (as a shorthand notation for the binary form the computer really uses) are what the computer understands most easily.

An additional problem is that the format that seems appropriate to the programmer may not seem so to the user. This may be due to the programmer's greater familiarity with the computer, or to the programmer's lack of familiarity with the subject of the application.

An example of another poorly thought out prompt comes, once again, from the local bank's automatic teller machine. When a customer wants to enter a deposit into the system, he or she must naturally enter the amount of the deposit by using a numeric keypad. On the machine at my local bank, the user is faced with the following prompt:

ENTER AMOUNT OF DEPOSIT: $0000.00

Ask yourself if there is anything that this prompt tells us about the format of the input. A range of valid values is implied, but is this range obvious to the casual user who is not used to computer technology?

I once stood behind a man who wanted to enter a deposit. The amount of his deposit was $143.58. So, when he was faced with the above prompt, he began by hitting the 1, intending to enter his amount left to right, the same order that we use when writing the amount. When he entered the 1, the display changed to:

ENTER AMOUNT OF DEPOSIT: $0000.01

The man looked at this display, shook his head, and hit the *clear* button. The amount he wanted to deposit didn't have a 1 in the cent position, it had an 8. The 1 was supposed to appear in the hundreds column. Figuring he had simply made a mistake, he went through the entire process again, and wound up with exactly the same display shown above, with the 1 still in the cent column.

Scratching his head, he decided that perhaps he was entering the numbers in the wrong order. So, starting over a second time, he entered an 8 instead of the 1 he previously had entered. Sure enough, the display put the 8 in the right spot, with the display showing:

ENTER AMOUNT OF DEPOSIT: $0000.08

He next entered the 5, figuring that this machine must prefer the numbers to be entered right to left, instead of left to right. However, when he did this, the display showed

ENTER AMOUNT OF DEPOSIT: $0000.85

instead of .58 as he had assumed it would. Now the man was totally confused, and was about ready to give up when I offered assistance. Being familiar with how programmers think, I spotted the mechanism being used with little difficulty.

The clever programmer knew that, since it was not possible to know where each digit would finally wind up once all the digits were input, he could use a mechanism of sliding the numbers from right to left in the display as they were entered. This would change the display from showing "$0000.00" to "$0000.01" when the 1 was entered, then to "$0000.14" when the 4 was entered, then to "$0001.43", "$0014.35", and finally to "$0143.58" when the 3, 5, and 8 were entered in order.

This clever programming trick confused the user to no end. It required that the user observe the changes that were occurring when he entered digits, and that he correctly interpret what those changes represented. This is entirely too much to expect from even an experienced user. A novice user has no hope.

ERROR MESSAGES Most systems are even more cryptic with their error messages than with prompt messages (if that is possible). In many systems, error messages use numbers instead of giving verbal information. The user is then required to look up the number of the message in a manual to find out what the message means. This is totally unacceptable in a modern system.

As much care should be taken in constructing meaningful, detailed error messages as is taken to prepare prompt messages. The message should explain as clearly as possible what the encountered error was (especially if the error was a direct result of a user input), what the user did wrong, and how the error can be corrected. In the case of fatal errors, the user must be told how to get back to a point just before the error occurred, and how to avoid the error a second time.

As in the case of prompt messages, error messages can become quite large. A message file can again be used in such a case. Note, however, that it is not a good idea to create a multilevel system like that used to give different prompts for novices and experts. Since errors are supposed to be exceptions, and not usual occurrences, we hope that the user won't see them very often. Therefore, it is unlikely that a user can become so familiar with a particular message that an abbreviated form would be understood.

Finally, some mechanism should be used to help the user to differentiate error messages from other types of text. Using color, reversed video (i.e., black characters on a white background, instead of the normal white characters on a black background), and flashing characters can all serve this purpose. Another idea is to set aside a particular area of the screen for displaying only error messages.

Dealing with the Screen

Most programming languages provide a simple method of output that allows the programmer to display a line of text on the screen. The position of that line is determined by the position of the screen cursor. The cursor remains at the position immediately following the last item that was output. When the screen is filled, the text on the screen **scrolls,** i.e., moves up so that the top line of the text leaves the screen and is lost forever. This makes room at the bottom of the screen for a new line of text. All new text appears at the bottom of the screen from this point on.

The characteristics of the hardware determine how many characters will fit on the screen at one time. This is determined by the maximum number of characters per line, and the number of lines that can be displayed. Most modern computers allow up to eighty characters on a line, although there are still many that use a screen width of fewer than forty characters. Some systems come with an option that allows the programmer to select either forty- or eighty-column displays.

The other dimension, the number of lines, is also not standardized. The range seems to be between twenty and twenty-five lines on most systems. Again, this is fixed by the hardware and cannot be modified by the programmer.

Unfortunately, the character and line restrictions are invariably machine specific, making a program that uses these features less **portable,** i.e., less easily moved from one manufacturer's computer to another's. However, it is possible to isolate these machine specific functions into independent modules. Then, when the system is transported from one computer to another, only these particular modules, and not the entire program, must be rewritten.

PAGING The first consideration is that scrolling is not a natural mechanism for users. It is found only in computer systems. In addition, it is often difficult for a user to detect a change on a screen that scrolls. A more natural method of presentation is to arrange information to be displayed in pages. A **page,** in this case, is exactly one screen of output. Users are accustomed to page formats because of their long experience with printed materials.

An entire screen should be displayed at one time whenever possible. This is especially true for long text output, such as instructions. A new page is displayed when the user is done with the current page. This can be handled with a simple mechanism that requires the user to hit a key in order to advance to the next page.

During the input phase of a program, displaying an entire screenful at once isn't always possible, since future inputs and their prompts may be determined by previous inputs. However, in such a case, simply use the bottom of the screen as a divider between one page and the next. Fill up the screen as normal, letting the lines accumulate down the screen. Eventually, the screen will be filled. When this happens, the next line would ordinarily cause the screen to scroll. Instead of letting this happen, erase the screen (most computers have a special instruction that performs this function). The next line would then be displayed at the top of the blank screen, thus becoming the first line on the next page.

Even better would be to use the **direct cursor control** mechanism that is available on most computers. In this case, the programmer uses a device to cause the cursor to go to a specific place on the screen. The screen is **addressed** by specifying, for instance, the row and column where the cursor should appear. Any output would then appear at this location. This is a most effective way of designing a form for input that requires a full page on the screen at once. There is no scrolling of the text, and the user can see the entire input form at one time.

CLUTTERED SCREENS It is important that the screen be easily read. On some systems this is easier to accomplish than on others, since some screens can hold more information. In addition, the physical size of the monitor used on a particular computer can affect the readability of the display, just as a smaller television screen size makes the picture less distinct.

A useful rule of thumb is to try not to cram as much text onto a screen as possible. Leave plenty of "white space," i.e., blank space, surrounding messages and input areas. Using a double space between lines of text makes it more readable, although obviously only half as much text can be displayed.

Remember that it doesn't make any difference to the user how many screens it takes to display the information or enter the inputs. Use the screen to group inputs into logical pages, including only related inputs on the same page. For instance, use one screen to enter the personal information about a customer (e.g., name, address, phone, etc.), and another screen to enter the information about purchases (item catalog number, item name, quantity, price, etc.). This makes it easier for the user to understand the expected inputs.

TIMING LOOPS One of the tricks that is often promoted by programming texts for beginners is the **timing loop.** This device is used to create a delay so that the user, supposedly, can read the text on the screen, and then automatically advance to the next screen. In BASIC, such a loop might look like:

```
FOR I = 1 TO 10000: NEXT I
```

This code does absolutely nothing except take time to perform. Such code inserted anywhere in the program will cause a delay for the length of time it takes the computer to run through the ten thousand iterations of the loop. While it might, at first glance, appear that this should happen quite quickly, in reality it requires several seconds to execute, thus creating a delay that is noticeable to the user.

This mechanism is to be avoided at all cost. First, it is very machine and program specific in terms of the length of the delay. The same loop just given may take ten seconds to perform on one system, but only one second on another. This means that the programmer has to know certain details about the hardware characteristics, something that is not reasonable. This type of loop also makes the program less portable.

Next, no matter how long the programmer makes the delay, it never seems to be long enough. First, users have varying reading speeds, so the time may be too short for a number of users, no matter how hard they try to read fast. Second, if the user is distracted for even an instant, the screen will probably change before he or she has had an opportunity to read it. Finally, even if the user has read the entire screen, he or she may have some difficulty understanding a portion of it and may wish to reread it. Using a time delay mechanism eliminates this possibility.

Rather than use such a device, it is far better to simply require the user to signal when he or she is done reading the page and ready to go on to the next page. You are probably already familiar with some systems that do this. In BASIC, the most frequently used method is to require the user to hit the return key in order to go on to the next page. A usual prompt message that appears at the bottom of the screen is:

```
HIT THE RETURN KEY TO CONTINUE:
```

A simple input and trap statement can be used to screen this input so that only the *return* key will cause the advance to the next page. For example:

```
610   INPUT ANSWER$
620   IF LEN(ANSWER$) <> 0   GOTO 610
```

This REPEAT-UNTIL loop will identify the *return* key by the fact that it is the only key that will produce a length of zero for a string variable input.

5 Program Testing and Debugging

5.1 INTRODUCTION

One of the most startling statistics about the programming function, even if you don't always believe statistics, is that the average amount of time spent in the testing and debugging phase of a project is between fifty percent and seventy-five percent of the entire project time! That statistic takes into account time spent designing, implementing, and documenting the system.

Why should that be? Is it that the testing and debugging tasks themselves are inherently difficult, or that they are made difficult by the complexity of the programs being tested/debugged? Are programs really so full of bugs? Or is it that so much emphasis is placed on guaranteeing a working system? The correct answer is "all of the above."

Programs are difficult to test and debug because they have a tendency to grow large and become complex. These same two reasons, in addition to the inherent difficulty of the testing and debugging tasks, are to blame for the large number of bugs in any sophisticated program. Finally, managers and users place enormous emphasis on some type of "proof" that the system will not cause uncorrectable errors during use. Unfortunately, this proof is not incontrovertable. One of the many inviolable laws about programming, known universally as Lubarsky's Law of Cybernetic Entomology, is "There is always one more bug."

We typically lump testing and debugging together. This is because the first leads naturally to the second. Indeed, in looking at the concept of testing itself, we find that the secret to successful testing is to think of it as a negative condition. The first impulse is to suggest that testing should try to prove the software correct. In other words, testing might be used to show the absence of bugs. This task turns out to be impractical at best, nearly impossible at worst. While computer scientists play around with mathematical "proofs" of program correctness, no easily applied technique has yet been constructed for such a purpose.

The other side of the coin, the negative, is to test for the *presence* of bugs. This seems a much more rewarding approach to the problem. For one thing, once a bug has been identified and located, we have some fairly well understood techniques for dealing with it. Locating the bugs, however, still remains the most difficult portion of the procedure.

We will take the following approach in discussing this situation. First, we will define several classes of errors that may occur within a program or system. You can't tell the players without a scoresheet. Once we know what to look for, we will begin the testing procedure with a technique known as **desk checking,** a euphemism for doing a lot of checking for errors by hand. This technique is not always efficient, but it can prove quite profitable in eradicating some types of bugs that would otherwise take a great deal of machine testing to find. One of the problems with this technique is that it is labor intensive. These days, human labor in the programming field is still more expensive than machine costs. However, using desk checking can actually decrease the amount of human time spent on testing and debugging further down the road.

Next, we'll look at the methods of designing **test schemes** that will give the system a good workout in the search for bugs. This can be tedious, but the object is to smoke out as many bugs as possible. The last thing any programmer wants is for his software's users to find bugs for him. To avoid this, a systemized plan must be developed for exercising the system as much as is practical. This should never be left to a kind of testing where various types of data are selected solely at random.

Finally, a number of the most useful debugging techniques from the toolboxes of professional programmers will be presented. These tools are used only to locate the source of the error. Any fix to the program will be entirely program dependent and so does not warrant discussing here.

Debugging is still something of a black art. Every programmer, no matter how good, has experienced the somewhat embarrassing and perplexing situation of having spent days looking for a bug, only to have a colleague walk into his office, peer over his shoulder at the program listing for a minute, tell him *exactly* where the bug is, and walk out.

Which brings us to the final point. If at all possible, have someone else test and debug your system. If the system was developed by several people, debug each other's portions. A fresh pair of eyes can make discovering a bug much easier. In many large computing shops, an entirely separate team of programmers is responsible for the quality control of all software developed. They create the test procedures and data, and either do the debugging themselves or return the system to the original programmer(s) for debugging. In addition, simply having other colleagues analyze your code, called **peer review,** can greatly enhance the testing phase of a project. Any technique that helps to identify bugs should be employed to ensure as accurate a system as possible.

5·2 TYPES OF ERRORS

When asked to define the jargon "bug," most novice programmers will be able to supply a quick response. This is usually along the lines of "an error during the execution of a program." While accurate to some extent, the "bug" in this definition is that it suggests there is only one type of bug, namely one that occurs when a program is running. Indeed, what the user sees is probably not even the bug itself, i.e., the source of the error, but the manifestation or visible evidence that a bug exists. While this manifestation may be an error message of some kind generated either by the software or the operating system, the message does not always point to the direct cause of the fault. In reality, the message will probably only point in the general direction of what went wrong.

The first problem is to decide just what *did* go wrong. The recognition of the various classes of errors that might occur makes this task much simpler. Since debugging generally works backward, from the appearance of a bug to its source, let's begin with looking at how bugs show up, then define what types of mistakes programmers make to create those bugs.

Manifestations of Bugs

Whenever I use the word "manifestation," I think immediately of one synonym, "ghost." In a way, it is most appropriate to call the appearance of a bug a ghost. First of all, the bug doesn't always make itself known until after it has done some kind of mischief. Second, when it does make an appearance, there is often no substance to the error that is described by the message generated; the actual error might bear no relationship whatsoever to what the message says. The bug therefore becomes elusive like a ghost, which one can (supposedly) put one's hand through. Finally, many types of bugs make very ghostly impressions by occasionally disappearing for long periods of time, only to crop up later at irregular intervals. One is never quite certain whether any debugging activity that was undertaken has completely eliminated a particular bug, or only made it go into hiding for a time.

There are three main ways that bugs make their presence known to us: compiler or interpreter checks, run-time errors, and invalid output.

Compiler Checks

Most compilers and interpreters will perform error checking on a program. These checks mainly have to do with whether the conventions of the particular language the compiler handles are followed. The most common example is a check for the proper syntax in every program statement. The **syntax** of a language is the set of grammatical rules that define how a valid "sentence" in that language is formed. For example, we know that

```
100 LET X = A * N / 5
```

is a valid statement in BASIC. It is not, however, a valid statement in Pascal. The statement would have to be written as

```
X := A * N / 5
```

in order for it to be recognized as valid by a Pascal compiler. If a "sentence" (i.e., statement) is not recognized by the compiler or interpreter, it is flagged as a syntax error.

Compilers are typically much more sophisticated at locating errors and potential errors in a program than are interpreters. This is because they translate an entire program into machine language at once. This means that they can look at groups of statements, not just one statement at a time. In addition, unlike interpreters, they can flag *all* syntax errors in the entire program at once. An interpreter can only identify one syntax error at a time, as it is encountered during the execution of the program.

About the best that the interpreter can provide is a very annoying "SYNTAX ERROR ON LINE xyz" message when it encounters something it does not understand. Unfortunately, the error is not always obvious, even though the user has been told what line it is on. More advanced interpreters will use some type of pointer to show where in the line it got confused, which at least gives you more of a starting point.

Compilers, on the other hand, typically will make a guess as to what type of error has occurred. While this may not always pinpoint the exact error, it is usually correct. For example, the UCSD (University of California at San Diego) Pascal compiler lists over 175 error messages that the compiler can generate.

Bugs encountered in the compilation stage are generally of two types, fatal and non-fatal. A fatal error means that the program definitely will not run until the error is corrected. An example might be when garbage is somehow inadvertently entered into the source code file. Until this garbage is removed, the program cannot be executed.

A non-fatal error is one that will not keep the program from executing, but that will probably result in erroneous results or a **run-time error** (a bug that occurs while the program is executing). Any statements that the compiler suspects will cause such problems are flagged with messages warning the programmer that something might go wrong. The programmer can choose to ignore the warning or to use it to look for a bug. An example of this type of error (for some compilers) would be declaring a variable twice.

Run-time Errors

Since interpreters execute a program statement immediately upon translating it into machine language, all errors identified by a language such as BASIC are actually run-time errors. The only technical exception to this might be the "SYNTAX ERROR," since this declares that a particular statement cannot actually be executed.

Run-time errors are much more difficult to deal with than those discovered during compiling. Compiler errors are usually fairly straightforward in their causes, and are likewise straightforward, for the most part, in their resolutions. On the other hand, run-time errors are much more devious in nature because of their unpredictability. One never knows where or when they will show up. And when they do finally make themselves known, about the only thing that can be safely assumed is that the source of the error is probably not anywhere near its eventual appearance.

Run-time errors can also be classified as fatal or non-fatal. Non-fatal errors can be corrected (usually by the user, during the execution) so that the program can continue. Most often, this error correction mechanism is a part of the software itself, as described in Chapter 4, Human-Engineered Programming. An example is an incorrectly formatted input. For instance, if the program was expecting a date in the format of "01/07/84" and the user entered "7 January 1984" instead, the program might ask the user to enter the date again.

Fatal errors halt the execution of the program as soon as the error is detected. If a program is running in a **batch** mode, i.e., has been given all its data and left to run by itself without further human intervention, this type of error could be generated either by the program itself or by the operating system. The example of the incorrectly formatted date above might prove fatal to a batch program if it couldn't figure out either how to correct or to ignore the faulty data, since there would not be a human user to correct the mistake.

In the case of **interactive** programs, ones that depend upon a human being to give them data from time to time, many types of data errors can be corrected by the user. However, there is still a multitude of errors that will cause the system to halt execution. For example, trying to execute the following BASIC statement

```
100 LET A = B / (D - E + 1)
```

when the variable D has the value 5 and the variable E has the value 6 could cause a "DIVIDE BY ZERO" error to occur. (Whether or not it does occur depends on the data types of the variables D and E, i.e., whether they are integers or floating point values.) Since the program may not be checking for such a condition, the operating system would step in and identify what caused the system to halt.

Note carefully the phrase "what caused the system to halt." This is all that can be identified by the operating system for run-time errors. The operating system cannot tell what caused the error to occur!

Invalid Output

Once you get to the point where compile and run-time errors don't plague you anymore, you can at last be more concerned with the accuracy of the output. The most obvious thing you will want to check for is whether the output is in the correct format. This correctness will be determined according to your (or your user's) view of how you want the output to look. Are the columns of the report properly aligned? Are the messages on the screen displayed in a readable format? Is everything exactly where it should be?

A much more subtle problem is determining if the actual data that is output is correctly calculated and of the proper accuracy. These things are much more difficult to detect and require careful testing. The test procedure includes using data that should produce known results. If the proper results are not generated by the program, then a bug has been discovered. However, the real danger is that the opposite is often assumed. It does not necessarily follow that if proper results *are* generated, then *no* bugs exist!

Errors of Design

Errors can invade a software project from the very beginning. If you recall the order of events, the first step in the process is the definition of the application, in consultation with the user(s). This requires the use of system design tools such as HIPO charts. The final step prior to implementing the system is the development of algorithms. While compile-time errors are mostly limited to syntax problems, most design errors manifest themselves as incorrect results.

Misinterpretation of User Needs

Communication skills vary widely among people, even those with similar backgrounds. So much of how people communicate is influenced by their personality, mood, and even physical motor skills. To complicate the process even more, the English language seems to be rampant with ambiguity, through the use of metaphors, similes, expressions, idioms, and colloquialisms. As if all that weren't bad enough, every special interest area creates its own vocabulary of jargon. It is no wonder, then, that so much *mis*-communication happens.

Even more confounding for the systems analyst is that users do not always know exactly what it is they want or need. This is especially true if the user has little experience with computers. It's the old situation of not knowing enough even to ask questions. This creates what is almost a trial-and-error condition for determining what the application is supposed to accomplish.

There is little you as a programmer can do when you show a user the new program you've developed for him and he says it doesn't do what he wants it to do, or doesn't do it in the way he wants it done. If things are handled properly, however, this situation probably will not occur. The idea is to keep the potential user informed all along the way of the design of the system. Even when you are personally both the user and the programmer of the system, unless you have a very clear idea of what you want the program to do, you might wind up creating a program that doesn't do the job you really need done. This is one of the main reasons you go through all the "hassle" of the formal design of the system in the first place.

Incorrect Use of Design Tools

The next possible entry point of errors into the system is in the tools and techniques that are employed during the design phase. In this case, a tool such as the HIPO chart might have been improperly used, making the design invalid.

Usually, however, any error in using a particular tool shows up only as a difficulty in using the tool, not in the results of the application of the tool. For instance, if you really don't understand how a HIPO chart works or how to create one properly, you probably won't be able to create one at all. In this case, more careful study of the technique is warranted.

Faulty Algorithm

The most deadly error of the design phase is the creation of an algorithm that has bugs. In this case, the bug has been *designed into* the overall system, rather than being associated with something external, such as a design tool. These are the most difficult bugs to track down and eliminate from a system.

The earlier bugs in an algorithm are discovered, the easier they will be to correct. Once these bugs are propagated by implementing them into code, they are substantially more difficult to find and fix.

For professional shops, this translates into great expense. The goal is to eliminate as many bugs as possible *before* the implementation process begins.

Errors of Implementation

Once there is some level of confidence that the design is correct, the system must be translated into code. While this translation should be quite straightforward, it is still very easy to introduce bugs during this process. On the bright side, however, bugs introduced here are much more easily corrected than those introduced during the design phase. This is because the implementation phase is a very well defined process, using languages with specific characteristics. Even at this stage, however, discovering that the bugs exist is no trivial matter.

Misinterpretation of the Design

The main reason that so many tools and techniques for designing software systems have been formalized is to help eliminate misinterpretation errors. If a programmer is given a set of design specifications in a standardized format, there should not be any room for ambiguity in the design itself. If the design tools are properly applied, then their meaning should be well understood. However, this assumes that the uses of the tools themselves are well understood by the programmer.

Certainly one of the measures of the competency of a programmer is that he or she understands design tools and techniques well enough to avoid misinterpreting the original design. Even when this is the case, however, human fallibility makes every human endeavor a risk to some degree. In an updated version of an old saying, "To err is human. To really foul things up takes a computer."

Syntax

As defined earlier, a programming language's **syntax** is the formal definition of how proper program statements are constructed. Syntax errors are probably the easiest of all program bugs to fix. They are often simply the result of mistyping a statement. The more devious sort result from a programmer's incomplete understanding of a programming language. However, this type of error is not only detectable by the machine itself, but some compilers even make suggestions as to how the problem can be corrected.

Faulty Implementation

This type of bug is second only to a faulty design in being difficult to detect and correct. The bug occurs when the logic used to implement a part of the program was incorrectly formed by the programmer. This is different from the misinterpretation of the design in that here the actual translation of the design to code was incorrectly handled.

For example, the programmer might simply forget to code a line or two of the design. This would certainly result in either a fatal crash of the system or erroneous output. Bugs introduced through a faulty implementation most often show up as run-time errors.

5·3 DESK CHECKING

While the first impulse might be to execute a program in order to begin testing it, this is actually not a very good idea. So many bugs probably exist in the code that some steps should first be taken to weed out at least the most "obvious."

Reading the code is the most fundamental tool of debugging; it will have to be done sooner or later. A careful reading of the algorithm and code can prove very fruitful. While this can obviously be an extremely time-consuming task, the human mind is still better than a machine at finding patterns, recognizing inconsistencies, and discovering illogic. Therefore, a systematic reading of the code before *any* formal tests take place will cut down on the number of necessary test runs. This ultimately saves both programmer time and computer time.

Next, "playing computer" by executing the algorithm or code by hand will help reveal logic errors. For algorithms, this is done when the algorithm is first designed, and for code, usually during the debugging phase. Hand executing the code can also be done to verify that the translation from algorithm to code has been done correctly.

Finally, compile-time errors should be corrected. This is a much more difficult problem when interpreters are used, however, since even syntax errors are identified only one at a time.

Planning the Hunt

The search for bugs should begin early in the life cycle of a program, while the bugs are still in their larval state, for a number of reasons. First, the sooner they are discovered, the less time the programmer has spent creating them. If the source of the error is the designer's misinterpretation of the needs of the user, then discovering a bug early can keep the programmer from spending too much time running down blind alleys. Second, the longer a bug remains in the system, the more it is cast in concrete, as the life cycle moves from specification to design to implementation, etc. Finally, bugs have a nasty habit of camouflaging themselves. The longer a bug remains in the system, the more difficult it is to see. In addition, the programmer becomes blind to bugs after a while. He or she builds up assumptions about the way a series of code works, for example, and never stops to analyze whether the code really does work that way.

Checking the Specifications

There is little the analyst can do to verify the correctness of the specifications himself, unless the system he is developing is for himself only. It is vital that the user be consulted on a regular basis about his or her needs. This includes going back to the user with the "final" specifications in order to allow him or her one last opportunity to make additions or modifications.

This final approval by the user is often accomplished by what is called a **walk-through.** This is a formal meeting between the analysts and the users where the analysts describe in detail exactly what the system will do and how it will do it. More informally, the lead analyst could meet with the main user to discuss the specifications that were developed, in order to verify that nothing was misinterpreted or left out.

Reading the Algorithm

Once the specifications have been verified, the analyst proceeds with the design phase, which leads to the development of algorithms for the logic of the system functions. Since the algorithms are the first representations of these functions and will form the basis for the implementation, it is important to verify that they do, in fact, perform the functions expected.

The first step is to read the algorithm (i.e., the pseudo-code), looking for clues to potential errors. The second step is to **hand execute** the algorithm, to pretend that *you* are the computer and to perform the actions indicated by each step of the algorithm using paper and pencil.

The main purpose of giving the algorithm a careful study is to look for "obvious" things that might cause an error. The following are the most obvious danger signs to beware:

1. *Variables never initialized:* All variables should be initialized explicitly in the program, even if the compiler being used does this automatically. Not all compilers initialize variables in exactly the same way. For instance, is a Boolean variable (such as in Pascal) initialized to TRUE or FALSE by the system?
2. *Variables used before being initialized:* This is a variation of (1) above. This problem arises when the code for initializing a variable comes *logically* (although perhaps not physically) after code which uses that variable.
3. *Improperly used constructs:* In this case, one of the fundamental control structures, such as the IF-THEN-ELSE statement in pseudo-code, has been incorrectly used or formed. The most notorious example of this is using a WHILE loop instead of a REPEAT-UNTIL loop (or vice versa). Another example is using a series of IF statements instead of a CASE statement.
4. *Overly complex constructs:* While not an error in itself, having statements embedded several levels deep in loops and IF statements should make one suspicious of the correctness of the logic. A rule of thumb is that no code should be more than three levels deep. (This is obviously not an absolute.) Consider rewriting any section of pseudo-code to simplify decision-making, such as by using a CASE statement instead of several IF statements. This type of problem will be more evident during the next phase of checking the algorithm, hand execution.
5. *Improper end-of-file testing:* This is a very common problem with programs that handle much file data, especially when dealing with more than one file at a time. A simple verification that at least the end-of-file condition has been tested for (even if not necessarily correctly) is a good idea. Again, this problem will become more obvious during hand execution.
6. *Missing "code":* Even when developing a seemingly simple program, it is fairly easy to miss something. A quick check to see that code for all the functions is present eliminates needless problems later on. It is more difficult to verify that *all* of the code needed is present within a particular routine, however. Again, this is best left to the next stage of testing.
7. *Unreachable "code":* It is important to verify that all conditions stated in control structures can occur, if only very seldom. A condition that is *always* false on an IF statement means either that

the condition was improperly stated or that there is no need for the IF construct at all. Check all conditions in loops, IF statements, and CASE statements to determine that they *can* have both true and false values.

8. *Illogical "code":* It is difficult to determine whether a particular piece of code "makes sense" simply by reading it. However, there are obvious things to look for in the way of garhage and statements that have no meaning. For instance, a statement that contains something like "X * 0.0" probahly has no logical meaning. Another example might be "LET X = X."

While simply reading the algorithm may seem an obvious and trivial exercise, this technique is not without merit. Indeed, it can help to eliminate many errors hefore they actually enter the system.

Hand Executing the Algorithm

The next step in verifying the algorithm is to hand execute it, using paper and pencil. By following through the steps of the algorithm, in the same way that a computer would execute code, you can discover much more ahout possible errors in the logic, especially missing code, unreachable code, and illogical code. While it won't be practical to give the algorithms the same level of workout that the program will receive during the formal testing of the system, hand executing is still a well-established tool used by professional programmers. Indeed, hand executing a program is sometimes the only way to find out exactly what is going on within the program.

The technique is fairly simple in definition but can be tedious in implementation. One must develop patience and diligence when employing it. The only things needed are listings of the algorithms, plenty of paper and sharp pencils (with decent erasers!), and perhaps a calculator for numeric work.

There is only one object: to "execute" the program using a small set of sample data, looking for sources of error. This is done by keeping track of the values of variables as you step through the algorithm. This is much easier when using actual code, hut can he done effectively with pseudo-code.

First, define some sample data to he used. If the algorithm calls for using files, make up a few representative examples of records for each file. Also, if the algorithm calls for data to be entered hy a user, have this written down so that it can he referred to consistently.

As with all testing, hand executing will probably be implemented more than once, so a standard set of test data can be helpful.

Next, write down the names of all variables used in the algorithm on one or more pieces of paper, leaving several lines underneath, or space to the side of, each variable for writing its value. Each time a value changes, cross out the old value and write the new value either below or beside the previous one. This is preferable to simply erasing the old value, as it allows you to go back to see the progression of the changes, perhaps making it possible to identify incorrect sequences of values.

Finally, begin "executing" the algorithm. If the statement calls for an input, take the data from the appropriate page you've prepared with sample data, noting that a value(s) has been "read." For output statements, again use a separate sheet for each file, and another for displays. You may not be able to represent the output exactly, such as in the case of graphics, but at least write down all text and numeric data that is output. At this stage, because you are dealing with pseudo-code, you don't have to worry too much about things like the exact location of items on the screen, or the exact record layout of a particular file.

Figure 5.3-1 gives a simple example of the hand execution of an algorithm that calculates the average for a specific number of values.

There are four items needed for conveniently hand executing an algorithm. The first item is the pseudo-code of the algorithm itself. The technique used is to keep a pointer at the statement that is currently being "executed." The pointer is moved after each instruction is completed.

The second item is a list of all the variables that are referenced in the algorithm. Whenever a variable is assigned a value, either with an assignment (:=) statement or with an INPUT statement, write down the new value for that variable. Then, whenever that variable is used later on in the code, use the value written in the list. Assignment statements, including all calculations of expressions, are "executed" by using the values given in this list, and writing down the new value for any variable that is changed by the statement.

The third item needed for hand execution is a blank sheet for writing down all of the output generated by the algorithm during its execution. This includes all OUTPUT statements that are encountered. Write down the messages and values that are output in their proper order; writing each output statement on a new line makes this easier to follow. Also include on this page all values that are entered in response to an INPUT statement.

```
Average.
BEGIN
  REPEAT
    OUTPUT "How many values?"
    INPUT N
    Total := 0;
    FOR Count := 1 TO N DO
    BEGIN
      OUTPUT "Enter a number: "
      INPUT Number                    ←————
      Total := Total + Number
    END;
    Avg := Total / N;
    OUTPUT "Average is ", Avg
    OUTPUT "Do you want to average another list?";
    INPUT Answer
  UNTIL Answer = "NO"
END.
```
a

N	Total	Count	Number	Avg
6	0	1	23	
	23	2	17	
	40	3	29	

b

FIGURE 5.3-1a and b: Four parts are used in testing an algorithm. The first is the pseudo-code of the algorithm, shown in (a) for the *Average* algorithm. A pointer is used to identify which line is currently being "executed." In this case, the INPUT statement is just being completed. The second part is a hand-kept list of the values of all the variables, as shown in (b). As the value of a variable changes, its previous value is crossed out. From these values, we can determine that we are in the third iteration of the loop (since *Count* is 3), and that there will be three more values input after the current one (since *N* is 6). This table also shows that the current input value (29) has not been added into the *Total* variable. The third part of hand-executing an algorithm is a list of the output generated by the algorithm. This is shown in (c). This shows that only three numbers have been input so far. The final part is a list of the test data, shown in part (d). In this case, values that have been input are crossed off to indicate that they have been "entered" into the algorithm.

This item is used primarily to verify the results of the execution of the algorithm. The output generated should be predictable, based on the set of test data that is input. In the example of Figure 5.3-1, the only inputs and outputs are to the user, not files. In the cases where files are referenced, keep a separate sheet for each file, inputting and outputting to this paper file as appropriate.

The final item used in this technique is the set of test data. On one or more sheets of paper, write down a small set of data that you

```
    OUTPUT


How many values?
6
Enter a number:
23
Enter a number:
17
Enter a number:
29
```

c

```
           TEST DATA


Number of values: 6

List of values:  23   17   29   19   30   24
```

d

FIGURE 5.3-1c and d

would like to use to test the algorithm. The data should be somewhat trivial, so that you can easily predict the output that the algorithm should generate. If you get the expected results, then the algorithm is probably sound enough to continue on to the implementation phase.

If you do not get the expected results, then you must search for an error in the algorithm. This search is performed by tracing back through the code, looking for where the value of a variable may have turned out wrong. This may have been caused by a manual error of writing down the incorrect value, or by a mistake in an instruction. Be sure that each value written down for a variable makes sense. This is one reason that it is helpful only to cross out—and not to erase—previous values of variables in the variable table.

In Figure 5.3-1, the execution has gone through a number of iterations of the main loop. Note that the variables in the table in (b) show the series of values each variable has had up to this point in the execution. Note that it has been necessary to provide somewhat more detail in the pseudo-code in terms of the variable names in order to make this practical.

It is also necessary to maintain some type of pointer to the step that is about to be executed. Here, the pointer shows that we are executing the *INPUT Number* statement. The only way to know exactly at what logical point we are in the execution, however, is to

Variable Table

N	Total	Count	Number	Avg
6	0	1	23	
	23	2	17	
	40	3	29	
	69	4	19	
	88	5	30	
	118	6	24	
	142	7		23.7
3	0	1	79	
	79	2	62	
	141	3	95	
	236	4		78.7

Output

```
How many values?
6
Enter a value:
23
Enter a value:
17
Enter a value:
29
Enter a value:
19
Enter a value:
30
Enter a value:
24
Average is  23.7
Do you want to average another list?
YES
How many values?
3
Enter a value:
79
Enter a value:
62
Enter a value:
95
Average is  78.7
Do you want to average another list?
NO
```

FIGURE 5.3-2: An example of a complete execution of the *Average* algorithm using two sets of values. It is simple to verify that the algorithm has produced the correct results by observing that the value left in the *Avg* variable is, indeed, the average of the list each time.

consult the variable table. In this case, we would need to look at the loop counter, *Count,* to see that we are in our third iteration of the loop. Consulting the variable *N* shows us that we need to perform three more iterations, including this one.

Finally, note that one variable, *Avg,* has nothing written underneath it. This is because that variable does not yet have a value. It would be erroneous to indicate an initial value (of, say, 0) unless a statement to this effect were executed. We should not assume that every variable is given an initial value. In this way, we can uncover two types of bugs described above, uninitialized or improperly initialized variables.

For instance, consider the variable *Total.* If the line initializing *Total* were removed, when it came time to add the first number into this accumulator, *Total* would not have the value zero. This could produce erroneous results. I say "could" because, like so much in programming, it depends on the context. If the code implemented from this algorithm were going to be executed only once, there would be no problem, as the variable would probably be automatically initialized by the compiler. But if the user wanted to average a second list, the total from the first list would be added to the total of the second. This is obviously incorrect.

Figure 5.3-2 shows the end results of testing the algorithm using two sets of numbers to be averaged. At this point, if you are satisfied that the outputs seem reasonable, you may assume that the algorithm functions essentially correctly. Obviously this is not absolute proof, but at least it covers the most blatant mistakes.

This type of testing should be done for all algorithms as soon as they are designed. It is seldom necessary to test them in this manner more than once or twice, since the actual code can be tested much more effectively, unless the algorithms have been modified in some way. In this case, testing should always start from the beginning again.

Comparing the Algorithm to the Code

The next step in desk checking is to compare the pseudo-code of the algorithms to the program code into which it was translated. There are two main purposes to this. First, it easily identifies missing code, if there are algorithms for which there is no code. Second, it can be used to help identify improperly used control structures. This is especially true when using languages such as BASIC which might not have the full range of control structures used in pseudo-code. An example would be the CASE statement. It is important to verify that any translation done from pseudo-code is handled with the utmost precision.

Reading the Code

The comparison above is mostly to verify the existence of code, not the effectiveness of it. Once it has been verified that, on the surface at least, the algorithm was translated appropriately into code, it is still necessary to verify the accuracy of that translation. This is done by first reading the code, searching for obvious mechanical mistakes, then hand executing the code, the same as is done for algorithms.

Because the program code, unlike pseudo-code, will be executed by machine, it is necessary to worry more about the precision of syntax and semantics. It is necessary to conform precisely to the syntactical conventions demanded by the particular programming language being used. The following are the main points to be concerned with while reading the code itself:

1. All points discussed above for reading the algorithm apply here as well.
2. *Syntax errors:* These take on a variety of guises. When using a good compiler, it is often simpler to let that tool ferret out any lapses in the use of syntax rules. However, when numerous errors occur within the same program, the compiler can often get confused. When using an interpreter, it is *not* a good idea to let the machine find syntax errors, because it won't find them until it tries to execute the errant line of code. This means that if a particular line of code had not been executed in any of the testing of the system (which could occur, especially in complex systems), it could hold a fatal trap for an unsuspecting user.

   Following are some of the more devious errors that are rarely trapped even by an intelligent compiler:

   a. *Improperly formed strings:* Usually this occurs when the closing quotation mark is inadvertently left off. This causes incredible havoc with the compiler, and usually causes very strange error messages that have absolutely nothing to do with the acual fault.
   b. *Improperly formed comments:* This is similar to (a). The worst case is a missing closing delimiter for a comment. What often

happens is that the compiler takes everything up to the *next* closing delimeter as being the comment. This means that numerous lines of legitimate code may be entirely ignored, without an error message even showing up in the compiler listing.

   c. *Missing block delimeters:* Missing a BEGIN or END indicator, or a RETURN for a subroutine, can change the logic of the program drastically. Again, the compiler may or may not be able to divine this particular problem. At best, it might indicate when a BEGIN is not properly matched with an END, although it won't have any idea *where* the END might belong.

   d. *Improperly formed loops:* The most common problem here is not adhering to the guidelines about the use of loops. For instance, having two loops overlap instead of one loop being entirely enclosed within another. Each loop should be checked to see that it does not violate any of the loop conventions discussed earlier (see section 3.2, Implementation Guidelines). Again, most compilers and interpreters can only spot the most blatant violations, and won't be able to pinpoint the problem.

   e. *Typos:* Garbage that would cause a syntax error should be fairly obvious to spot and correct.

3. *Bad variables:* There are a number of things that can go wrong in the seemingly simple use of variables. The three things important about variables are name, type, and value.

   a. *Misspelled variable name:* This can often be difficult to find when just reading the code, unless the misspelling is quite obvious. This is especially true for languages such as BASIC that do not require explicit variable declarations. BASIC itself could never find a misspelled name. Cross-reference tables are often useful for this type of check (see section 5.5, Debugging Techniques).

   b. *Violation of variable naming conventions:* Any restrictions on variable names can cause trouble. There are usually length restrictions, limited characters that may be used in names, or particular **reserved words** that may not be used as variable names. In addition, if you are using any of your own conventions, such as those suggested in section 3.4, Effective Use of Subroutines, be certain that all variables conform.

   c. *Undeclared or improperly declared variables:* While languages such as Pascal require that all variables be declared explicitly, BASIC has little ability for declarations. However, whatever requirements there are in the language, make certain they are followed precisely, and that variables are declared with proper types and value ranges.

d. *Variable type incompatibilities:* Look for statements that use variables of different types. Make certain that conversions from one type to another (as from string to numeric) are handled properly, using the appropriate functions for such conversions.
e. *Values out of range:* Where variables may have only a specific range of values, look for violations of this range. This is usually better identified through the hand execution technique.

4. *Illegal array subscripts:* This is similar to (3e) above, in that this type of error usually manifests itself as a value out of range. For instance, using a negative value as a subscript in BASIC would be an error. The proper range of values for subscripts in BASIC is usually integers between 0 and the upper limit that was specified in the DIM statement for that array. However, in some systems, the value 0 might be invalid. In Pascal, any range of values, including negatives and nonnumeric values, can be made valid by using the proper declaration.
5. *Improper file use:* There are three main problems that occur along this line:

   a. *Improper file access:* Here the conventions for accessing the file using special statements (such as the OPEN, CLOSE, READ, or WRITE statements in BASIC) have been violated. This is often a syntax problem, but may be caused by using the wrong type of statement (such as using the statement for reading a random-access file on a sequential file) or missing a statement altogether.
   b. *Inconsistent record layouts:* In this case, the record layout for the file being used is incorrect and does not correspond to the way data is actually stored in the file. This will probably not be immediately obvious if the file in question is an output file that some other system later uses as an input file. The file formats used should be carefully verified with the file definitions given in the specifications.
   c. *Improper end-of-file testing:* As noted in section 3.2, Implementation Guidelines, care must be taken to take into account any contingencies for reaching the end of a file.

6. *Variables initialized but never used:* While not an error in itself, a variable such as this can cause a great deal of confusion for anyone reading the program for maintenance purposes. In addition, finding a variable of this type might point to a piece of logic that was never implemented. At the least, a declared and initialized variable that is never used wastes time and space, both of which might be in short supply.
7. *Violation of subroutine conventions:* It is important to maintain consistent use of subroutines, especially in a language such as BASIC, even through artificial conventions. This was pointed out in section 3.4, Effective Use of Subroutines.
8. *Inaccurate calculations:* This is one of the most difficult areas to judge just by reading the code. The problems of tolerances, the inexactness of internal representations, round-off and truncation errors, and the use of approximations all conspire to make any attempt at accuracy a hoax. Conversions and comparisons between integers and floating point values are the most notorious offenders of this point.

Hand Executing the Code

There is really little difference between the technique of hand executing an algorithm and hand executing code. The principle and goals are the same. It might be easier to apply this technique to code, however, because the necessary actions are explicitly stated, rather than implied, as with pseudo-code. The main differences when dealing with code would be the need to keep track of more variables, and to be more precise when dealing with files. It will still be necessary, however, to ignore some statements that cannot be readily duplicated using paper and pencil.

Hand executing code is used most frequently during the actual testing phase, where it is employed to aid the debugging process. It remains one of the time-honored methods for tracking down the sources of bugs in a system.

5·4 DESIGNING TEST SCHEMES

Professional programmers often have sophisticated systems available for developing sets of test data that will help them to test the correctness of new systems. These tools include systems both to generate and to maintain test data in large libraries. In this way, the data can be used over again when the system needs to be tested in the future (when there has been a modification to the system). This is all especially helpful when dealing with rather large systems, since the volume of test data needed can indeed be massive.

Unfortunately, such test-generation systems are themselves extremely large and expensive. It is unlikely that systems of this type will be available for microcomputers in the near future. In addition, it is not entirely clear that the systems do, in fact, produce better sets of data with which to perform tests than do human beings. More likely, this type of activity requires the peculiarities of the human mind to be done well. A great deal of creativity must be used in designing appropriate test data.

In the end, most programmers wind up creating their own sets of test data for any new program they write. This, too, is not necessarily desirable. If the data used to test the system was not prepared carefully, much mayhem could result in the future when the system is being used to do real work. Human beings are notorious for missing important details. Fortunately, there are some basic guidelines that can be followed so that chances of successful testing are rather good.

Smoking Out the Bugs

The first difficulty in designing test data is in defining just what a "successful test" really is. On first thought, it might seem that a test for a new system would be successful if the system performed as expected, doing the job it was supposed to do without any kind of failure. Since it is rather easy to equate "success" with "no errors," it is also easy to accept this definition.

If we *do* accept this definition, the quickest and simplest way to test the system would be to design test data that would be considered normal. If executing the new system using this test data produced no errors, we could suppose that the system worked.

Unfortunately, life is never that simple. Proving the correctness of a program is impossible as a practical matter. Having a program execute correctly with a particular, representative set of test data is of relatively little importance, since the program might fail if given a different set of valid data.

The test phase should not be approached as an effort to prove that the program is correct, at least not directly. Rather, the testing should be directed at trying to make the program *fail*. By using the deviousness of the human mind to attempt to crash the program, the tester will subject the program to as rigorous a test as it might ever face during its real work life. The object is to provide a program that a *user* cannot crash. Logic indicates that a program that the *designer* cannot crash must indeed be "error free," as far as the user is concerned.

This is the same principle used by test pilots of airplanes. First, test pilots are often not only pilots, but also engineers. These engineer-pilots take planes to their outermost limits, looking for defects in the design. It would be much easier to fly the planes as they would normally be flown. But the object of the test is to find the weak points of the plane, not simply to determine that the plane will fly. Only when those weak points and limits of operation are discovered is the test successful. The weak points can then be reengineered in an effort to eliminate them, and the limits can either be accepted or also reengineered.

The object of testing a program, then, is to find its limitations and weak points, which usually manifest themselves as bugs. A well-organized plan of attack can be used to cause those bugs to show up where they will do the least harm—in the test phase. If the program shows errors during testing, this is not an indication that the programmer has failed. It has been said that every program ever written (above a certain threshold of complexity) will have at least one bug in it (recall Lubarsky's Law). That particular bug might be somewhat inconsequential (such as a comma out of place), but it still exists. Bugs in new software are an accepted fact of the computer business. This does not mean that bugs are to be tolerated. Rather it is an indication of the difficulty of developing bug-free software.

The design and implementation steps taken up to now have primarily been an effort to allow as few bugs into the system as possible, and to minimize their effects. The test phase is the last line of defense against this bane of all programmers.

Once testing produces a bug, various debugging techniques (as discussed in the next section) can be employed to locate and eradicate it. This testing and debugging is an iterative process, since seldom can more than one bug at a time be made visible during a particular test. Eventually, the program will pass all the tests correctly. At this point, the system is as error-free as it can be made for the present, and is therefore made available to users.

The end of testing does not mean, however, that the system does not have any bugs in it. It simply means that all bugs that were discovered during testing were eliminated. The need for rigorous testing should be quite evident. Unless the software is thoroughly tested, there can be little confidence that (nearly) all of the bugs have been eliminated.

Following are some of the techniques that are regularly used by professional programmers to provide thorough testing of a new program.

Test Schemes

A **test scheme** is a strategy or collection of strategies for testing a set of code, whether at the module, program, or system level. Setting up a test scheme involves devising one or more **test cases,** sets of input values that can be used to obtain information about the inner workings of the code and to discover any lurking bugs.

A common error made by testers is to give the program what is often called a "smoke test" by entering random data and looking for any "smoke" that might develop during the program execution, signaling an imminent crash. (This type of testing undoubtedly gave rise to the mythical HCF assembly language instruction, Halt and Catch Fire!) While this type of testing will probably indeed identify errors in the code, these will be only the most obvious errors. You can be sure that the more devious types are still lurking.

Another problem with using purely random data is that it is difficult at best to predict the output a module or program should produce using the data. This makes evaluating the test results additionally complex, wasting time and introducing yet another potential source of error—misinterpreting test results.

Exhaustive Testing

It has been theorized that there is only one certain way to test a program. Every possible combination of every input must be used to test the program. Since this **exhaustive input** test itself would cause the program to execute in every way that is possible for it to be executed, using every possible value for every input, such a test could "prove" that a program was correct. In this way, every possible set of outputs could be generated by the program, which should be sufficient for discovering errors. Unfortunately, this approach has two serious flaws. First, such a test would not necessarily point out whether any code were missing, but would instead only point out bugs in what code is present. Second, and more important, is that this approach is not practical except in the case of the most trivial programs. To exhaustively test even a small program in this fashion could take billions of years!

Another type of exhaustive testing, called **exhaustive path** testing, calls for creating test cases that would execute every possible path within a program. In this way, it is hoped, since every line of code would be executed in every possible combination, all bugs would be forced to surface.

While the number of possible paths through any program is undoubtedly much smaller than the number of test cases required in an exhaustive input test, the number is still large enough to make exhaustive path testing impossible on a practical level. In addition, there are types of errors that can occur within a program that are not detectable using this type of test.

Developing Test Cases

While it is not possible to test every possible path through a program, it is always a good idea to cause every possible line of code to execute at least once. In interpreted languages such as BASIC, this is essential, since there may be a syntax error remaining in a line of code that has never been executed in a test. Recall that syntax errors in BASIC are not identified until the line of code with the error is *executed.*

The remaining discussion revolves around creating test cases that will force the execution through every line of code at least once. In addition, the logic of the program, controlled with conditions in statements such as IF, WHILE, and FOR, must be exercised as much as possible.

Test cases are usually devised for each level of testing, even hand execution of the algorithm. In the early stages of testing, the test data is usually informally defined, and will not test the system as rigorously. The main purpose of this stage is to test the main logic of the program. Later, additional data can be added to make the testing more complete. In addition, as will be seen, part of the object of testing is to verify any error-detecting done by your program itself.

While there are no formal rules that can be followed in creating test data (if there were, the task would be completely automated!), there are some heuristics (rules of thumb) that can be followed to help to maximize the testing effort.

Test cases are devised with three thoughts in mind. The first is that we want to force every line of code to execute at least once. This will mean, in a well-formed program with many user friendly devices, that the test data must force execution down logic paths that we hope the user will never take. An example is an error routine. Therefore, the test data must contain data that the program will consider invalid.

Second, we must consider where in the logic errors are likely to occur and test that code thoroughly. As mentioned above, the most frequent errors occur wherever conditions are used to control the flow through the program, i.e., in IF statements, WHILE statements, and the like. Any statement that contains a condition test or comparison is suspect for error.

Finally, we are interested in producing regular output from the program with the tests. This is the only way that the main logic can be verified. The output is examined for any irregularities or inaccuracies. This means that the results should be anticipated before the test so that the actual output of the test can be compared to the expected output. In addition, especially if this program is being developed for someone else, this gives the user a final opportunity to make certain that the program does the job he or she needs it to do.

It should be obvious by now that testing a program is not a trivial matter. In addition, we are not talking about a single, once-and-done type of test, but rather a series of tests using different sets of test data designed with specific results in mind. These tests may also need to be repeated whenever the program being tested is patched or has code added. Much care should go into setting up the test data in order to achieve the maximum benefit.

Normal Cases

The first class of test cases is the set containing what one would consider normal data. This data is used to verify the main logic of the program, producing output that can be easily verified. The data does not need to be extensive but should be complex enough to cause calculations and decisions that are something more than trivial. However, keeping this data simple is useful, since the results will need to be verified by hand.

It might seem obvious how to go about constructing valid test data, but to avoid mistakes, it is helpful to keep in mind that data has three main attributes: type, value, and format. The format is often related to type and value, and is certainly most important when dealing with input data.

Type is most often concerned with whether the variable's value is maintained in internal format as a character, integer, or floating point number. Other types exist in some languages. In fact, in Pascal it is possible for a programmer to define his or her own types of data. The most common error for this attribute is what is often called a **type mismatch,** where one type of value is assigned to a variable of a different type. For example, if a numeric value is assigned to a string variable in BASIC (LET A$ = 3), the program halts. Another potential problem with mismatched types such as with integers and floating point numbers, is one of the precision of the result of a calculation. In this case, truncation or rounding can cause what would be considered erroneous results.

Value has to do with what values it is valid for a variable to hold. This most often shows up as a specific range. For instance, if we're interested in a variable that represents months as numbers, we would expect this variable to have an integer value between 1 and 12.

Finally, the format of a variable has to do with any specific way a value is represented that is not one of the standard representations, such as character or integer type. In this case, we might be interested in creating our own requirements for the variable's type. For instance, consider a date. There are many ways that a date can be represented. A particular program might require the date to appear as "MO/DA/YR", where MO is a two-digit number to represent a month, DA is a two-digit date, and YR is the last two digits of the current year.

Valid or normal data could then be defined as any data that meets the requirements of type, value, and format for variables to which it is assigned.

Consider the program given in Figure 5.4-1, a simple program for calculating student grades. The program requests two categories of input. The first is the number of grades to be entered. The second is a group of grades, one at a time. Normally, the program specifications and the data definitions are consulted to determine what data would be valid for this program. In this case, we can see from the code that the value of NGRADES, the variable for the number of grades, must be an integer between 0 and 25, inclusive.

The second category of input is in the form of GRTEMP, a temporary variable holding a grade that is being added to the array GRADE. In this case, the value must be a number between 0 and 100, inclusive.

It would be appropriate to set up valid test data, run the program by inputting this data, then evaulate the results for any format or calculation errors. For instance, inputting 4 for the number of grades and 79.8, 82.6, 94.5, and 78.6 for the grades should produce a raw grade of 83.875 and a final grade of "B." If it doesn't, something is wrong with the program.

```
100 DIM GRADE(25)
110 PRINT
120 PRINT
130 INPUT "How many grades?  ",NGRADES
140 IF (NGRADES < 1) OR (NGRADES > 25) THEN
       PRINT "*** Must be between 1 and 25, inclusive ***":
       GOTO 130
150 GRTOTAL = 0
160 PRINT
170 FOR I = 1 TO NGRADES
180   PRINT
190   PRINT "Enter grade #";I;":  ";
200   INPUT " ",GRTEMP
210   IF (GRTEMP < 0) OR (GRTEMP > 100) THEN
         PRINT "*** invalid grade -- must be between 0 and 100";:
         PRINT ", inclusive ***":
         GOTO 190
220   GRADE(I) = GRTEMP
230   GRTOTAL = GRTOTAL + GRTEMP
240 NEXT I
250 RAWGRADE = GRTOTAL / NGRADES
260 IF RAWGRADE >= 90 THEN
       FINALGRADE$ = "A":
       GOTO 310
270 IF (RAWGRADE < 90) AND (RAWGRADE >= 80) THEN
       FINALGRADE$ = "B":
       GOTO 310
280 IF (RAWGRADE < 80) AND (RAWGRADE >= 70) THEN
       FINALGRADE$ = "C":
       GOTO 310
290 IF (RAWGRADE < 70) AND (RAWGRADE >= 60) THEN
       FINALGRADE$ = "D":
       GOTO 310
300 FINALGRADE$ = "F"
310 PRINT
320 PRINT
330 PRINT "******************************************"
340 PRINT
350 PRINT "The student's raw grade of ";RAWGRADE;"% is a letter";
360 PRINT " grade of ";FINALGRADE$
370 PRINT
380 INPUT "Do you want to calculate another grade (Y/N)?  ",ANSWER$
390 IF (ANSWER$ =  "Y") OR (ANSWER$ = "y") GOTO 110
999 END
```

FIGURE 5.4-1: A simple BASIC program to calculate a student's grade.

Exceptional Cases

An exceptional test case is where the data, while valid, represents an unusual twist for the program. This is somewhat hard to define, since not all programs have logic that make this type of test obvious. Figure 5.4-2 gives code that will calculate the roots of a quadratic

```
100 PRINT "Enter values A, B, and C"
105 PRINT "for the quadratic equation"
110 PRINT
120 PRINT "A*x*x + B*x + C = 0"
130 PRINT
140 INPUT A, B, C
150 TERM = SQR(B*B - 4*A*C)
160 X1 = (TERM - B) / (2 * A)
170 X2 = (-TERM - B) / (2 * A)
180 PRINT "Root #1 is "; X1
190 PRINT "Root #2 is "; X2
999 END
```

FIGURE 5.4-2: A BASIC program that calculates the roots of a quadratic equation. What kind of data should be used to test this program?

equation using the quadratic formula. A normal test case would be (1, 2, 1). An exceptional case might be (10, 0, 0). This is exceptional because it causes the B and C terms to drop out of the equation resulting in the calculation evaluating to zero. However, the code for calculating the quadratic formula should be able to handle the situation. In addition, the data (0, 10, 7) is also mathematically valid, resulting in the equation (10x + 7 = 0). Notice that the code will not satisfactorily handle this data, even though it is technically valid. A final set of valid values might be (0, 0, 0).

Exceptional values often take on the form of extremes, forcing values to their extreme highs and lows, to observe the results. For instance, in the grade program above, the tester should force the average to be very high and very low in tests. Using large and small numbers can especially point out any problems in formats, such as those used in a PRINT USING statement in BASIC.

Creating exceptional test data is an attempt to force the program to do something that might be unexpected. Using extreme values is about the only way to discover certain unexpected problems.

Boundary Cases

Boundary cases attempt to create data that will force evaluation of conditions and comparisons into boundary values, i.e., values that lie right on the border between valid and invalid. This is done mainly to locate errors in comparisons. A classic example is the "off by one" error, where a counter's value, such as a loop control variable, is either one more or one less than it should. This can result from initializing the loop control variable or counter incorrectly, from setting up the loop exit comparison incorrectly, or from using the wrong kind of loop (e.g., a WHILE instead of a REPEAT-UNTIL).

The main goal is to verify that the range of a value is properly maintained, and that the lowest and highest values in the range are indeed valid. For instance, in the grade program, the number of grades can be between 0 and 25, inclusive. Boundary tests would select both of these values. In fact, one of these values causes an error in the program. It is left as an exercise to discover which one.

Other boundary conditions that should be tested are the range of grade values (0 to 100, inclusive), and the final grade determinations. Test data should be devised that test the boundary raw scores of 90, 80, 70, and 60. In addition, raw scores of 0 and 100 should be tested.

Another source of boundary problems is in dealing with empty files. While it might seem somewhat ridiculous to expect a program to run without an input file, there are times when this legitimately happens. Many programs might need to be executed even if some input file is empty. Consider, for instance, a program that takes the number of sales for a given month as input and reports on monthly sales information. What if there were no sales that month? Certainly not a reassuring prospect for the company, but the software should be able to handle this situation. For instance, what if we are talking about a real estate office instead of, say, an auto sales company. Now the possibility of zero sales in a month seems more understandable.

Finally, another boundary worth checking is one that reveals what happens when there is exactly one record in a file.

Invalid Cases

Since every program should be designed to detect certain classes of error, test data should be developed that will force the program to prove that it can, in fact, detect those errors. This test data should either represent input that is entirely out of acceptable bounds, or produce calculations that will go out of bounds. In addition, however, the data should attempt to cause the program to crash! By making up weird input, you hope to create a type of error that the programmer or designer did not anticipate. Unfortunately, as in previous cases, there is no well-established way to do this. After a while, however, you get a certain "sixth sense" about such things.

For the grade program, invalid data would include any number less than 1 or greater than 25 for the number of grades. Other invalid data would be any grade not within the proper range.

More difficult than invalid values is trying to force calculated values to be out of range. In this example, the raw grade could cause a problem if its value did not fall within the range of 0 to 100, inclusive. Can you devise input data that will force the variable RAWGRADE outside of this range? If not, why not? (The answer has to do with how values are input and verified.)

Non-incremental Testing

The most obvious time to begin testing a program is after all the modules have been coded. It would then be natural to merge all of the modules and test them as a whole. Myers [77] calls this the "big bang" approach. Its main appeal is that it appears to save time during the test phase of the project.

As we'll see in the next section, however, testing the whole program at once makes the task much more difficult. First, making up the test data is more complex because the entire set of data must be devised and properly set up at once. Second, any problems that develop during "big bang" testing are much more difficult to locate and correct, mainly due to the size of the code being tested. Finally, it is more difficult to verify that the bug has been eliminated, and not just forced somewhere else, when there is so much code being tested.

For these reasons, this approach, although widely used, is not considered highly reliable. Eventually any program must be tested as a whole. However, this should only be done after incremental tests are performed. In this way, the confidence in the modules, and thus the entire program, is increased to the highest possible level.

Incremental Testing

As mentioned in section 1.3, Modular Design, one of the purposes of a modular division of program functions is to facilitate testing the program. This is done through the **incremental testing** of each module. In this method, each of the modules is initially tested by itself.

This type of testing helps in three ways. First, developing the test data for the modules is easier than developing it for the entire program at once. Second, it tends to isolate the bugs within individual modules. Finally, the testing can be done as each module is implemented without waiting for other, unrelated modules to be implemented. This allows implementation, testing, and debugging to be conducted in a coordinated manner, operating on a single module or small group of modules at one time.

This method is not without its difficulties, however. The first difficulty is that it requires extra coding. The nature of that extra coding depends on whether implementation of the modules is being done using a top-down or bottom-up technique.

Top-Down Testing

Figure 5.4-3 gives the specifications and an H-chart for a program that will balance a checking account. As mentioned in previous chapters, we have a choice to make about how to proceed with the design and implementation of this program—top-down, bottom-up, or using some combination of the two. We have a similar choice when deciding how modules are going to be tested. The incremental testing methods follow the same philosophy of their design and implementation counterparts. If implementation was done top-down, then it would be most natural for the testing also to follow this method. This is especially true if the modules are to be tested as they are implemented, without waiting for lower-level modules to be coded.

Consider the checkbook program. In a top-down implementation, the highest level module (*Checkbook*) would be implemented first (see Figure 5.4-4). Therefore, with top-down incremental testing, that module would be immediately tested before the lower-level modules (*Enter-Checks, Verify-Checks, Enter-Deposits*, etc.) were coded.

This presents some interesting problems for the test. First, how can the *Checkbook* module be tested if other modules that are to be called by *Checkbook* are not yet coded? The module being tested must make appropriate calls to whatever modules it must use. Otherwise it is not fully implemented and should not be tested.

The usual mechanism to handle this is to create what is called a **module stub** or **dummy routine.** This stub is a short piece of code that simulates the interaction of the calling routine and the called subroutine. This simulation is usually nothing sophisticated, since anything too elaborate would defeat the purpose of the technique. The objective is to test the main module, not the subroutine. The stub must be trivial in nature so as not to provide any opportunity for introducing a bug.

This is often accomplished by creating a stub that does nothing but accept the parameters passed. The stub would then output these

```
Program Name:  Checkbook

Specifications:

    This program is to help a user balance his or her checkbook
by performing the calculations required each month when the user
receives his or her statement.  The user should be allowed to
enter up to one hundred outstanding checks and up to twenty
outstanding deposits.  The user should be allowed to review the
checks and deposits entered, in case a mistake was made
entering them.  The user should then enter the balance from the
previous statement, and the last amount written in his or her
check register.  The program should calculate a new balance, and
output a report.  This report should contain a list of all the
outstanding checks and deposits, the previous balance, the total
amount for all outstanding checks and deposits, the last amount
written in the user's check register, and the new balance.  The
program should also output a statement of whether the checkbook
is balanced, and whether the account is overdrawn.  If the
checkbook does not balance, then the user has made one or more of
the following mistakes: (1) he or she has entered one or more
erroneous amounts for checks, deposits, the previous balance, or
the last amount written in the check register; (2) he or she has
included a check or deposit in the list of those outstanding when
that check or deposit is not outstanding; (3) he or she has not
included a check or deposit in the list of those outstanding
when, in fact, it is outstanding; (4) the user made an arithmetic
error in the check register, which made the final amount written
in the register incorrect.  In any case, the user should have an
opportunity to modify any of the values and get a new report.
```

FIGURE 5.4-3a: The definition of a program to balance a checkbook, including the specifications (a) and H-chart of the modules (b).

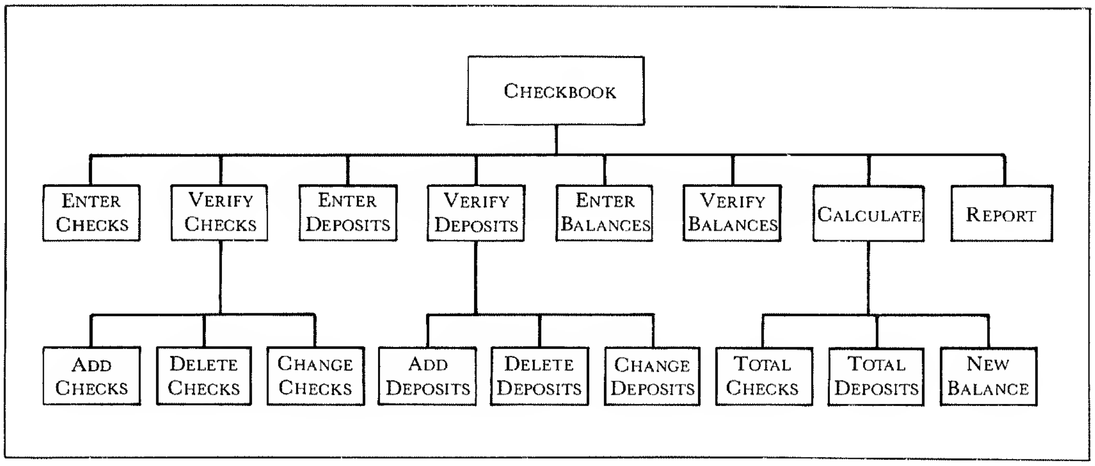


FIGURE 5.4-3b

parameters to a printer, allowing the tester to see how well the interface from the main module to the subroutine actually worked. Even simpler is a stub that only prints out its name, indicating that it has been called. Figure 5.4-5 gives examples of such stubs.

If the main program is expecting some output from the subroutine, it will be necessary to use test data to set up these output parameters within the subroutine. As will be evident after we discuss the creation of test data later in this chapter, this is not necessarily simple, since the stub cannot easily change this test data for multiple calls to the subroutine stub. For instance, in Figure 5.4-6 we have a program that calculates the average of a list of numbers. The program has been implemented from the top down, so we would like to test the main routine first. This requires that the subroutine *Enter-Values* be simulated using a test stub. In this case, all that is required is to set up the output parameters of the *Enter-Values* module with test data. The main program will then use these test values in its own calculations. In this example, the variables *N* and *TOTAL* must be supplied with values by the test stub in order for the main program to calculate the average in line 150.

Because the test data is **hard coded** into the test stubs using literal values and assignment statements, it is difficult to create a test stub that provides different values for its output parameters each time it is called. For instance, in the averaging program, the results are the same no matter how many times the tester responds "Y" when asked if he or she would like to average another list of numbers. In the

```
100 DIM CHECK(100), CHECKNUM$(100), DEPOSIT(20)
110 PRINT
120 PRINT
130 PRINT "This program helps you balance your checkbook.  You will"
140 PRINT "be asked to enter all of your outstanding checks and"
150 PRINT "deposits.  You must also enter the balance from your"
160 PRINT "previous statement.  Finally, you will enter the last"
170 PRINT "amount written in your checkbook register.  The program"
180 PRINT "will then tell you if your account is balanced."
190 PRINT
200 PRINT
210 '
220 ' Initialization section
230 '
240 CLEAR        'sets all numeric values to 0, all strings to null
250 OPTION BASE 1
260 '
270 PRINT "******************************************"
280 FORMAT1$ = "\      \ $$######.##   "
290 FORMAT2$ = "$$######.##"
300 FORMAT3$ = "##   $$######.##     "
310 '
320 ' end initialization
330 '
340 GOSUB 1010                              'Enter-Checks
350 GOSUB 1310                              'Verify-Checks
360 GOSUB 1610                              'Enter-Deposits
370 GOSUB 1910                              'Verify-Deposits
380 GOSUB 2210                              'Enter-Balances
390 GOSUB 2510                              'Verify-Balances
400 GOSUB 2810                              'Calculate
410 GOSUB 3110                              'Report
420 '
430 ' Create a menu in case the user wants to change some things
440 '
450 PRINT
460 PRINT
470 PRINT "Enter the number of your choice."
480 PRINT
490 PRINT
500 PRINT "1  --  Edit the list of checks"
510 PRINT
520 PRINT "2  --  Edit the list of deposits"
530 PRINT
540 PRINT "3  --  Edit the previous balance or register amount"
550 PRINT
560 PRINT "4  --  Output the report"
570 PRINT
580 PRINT "5  --  Start all over from the beginning"
590 PRINT
600 PRINT "6  --  Stop"
610 PRINT
```

FIGURE 5.4-4: The main module for the *Checkbook* program. In a top-down implementation, the main module is coded and tested before any other modules are implemented. This requires that stubs be created for each sub-module of the system. See Figure 5.4-5 for an example of stubs for this program.

```
620 '
630 INPUT ANSWER$
640 CHOICE% = INT(VAL(ANSWER$))    'convert string to an integer
650 IF (CHOICE% < 1) OR (CHOICE% > 6) THEN
         PRINT "***** Invalid entry, please reenter *****":
         GOTO 450
660 '
670 ' 1 -- execute Verify-Checks routine
680 '
690 IF CHOICE% = 1 THEN
         GOSUB 1310:
         GOTO 450
700 '
710 ' 2 -- execute Verify-Deposits routine
720 '
730 IF CHOICE% = 2 THEN
         GOSUB 1910:
         GOTO 450
740 '
750 ' 3 -- execute Verify-Balances routine
760 '
770 IF CHOICE% = 3 THEN
         GOSUB 2510:
         GOTO 450
780 '
790 ' 4 -- execute Calculate and Report routines
800 '
810 IF CHOICE% = 4 THEN
         GOSUB 2810:
         GOSUB 3110:
         GOTO 450
820 '
830 ' 5 -- start over, return to beginning of program
840 '
850 IF CHOICE% = 5 GOTO 110
860 '
870 ' 6 -- stop by going to end of program
880 '
890 PRINT
900 PRINT "***** End of Program *****"
910 PRINT
920 GOTO 9999
9999 END
```

FIGURE 5.4-4 ***continued***

checkbook program of Figures 5.4-4 and 5.4-5, how can the test stub for the *Verify-Checks* module provide different results when the subroutine is called from lines 350 and 690?

In the case of BASIC, it might be possible to use READ and DATA statements to provide such a mechanism. Each time the stub is called, it could READ a different set of test data to be returned

```
1000 '
1001 ' Enter-Checks
1002 '
1010 PRINT "Enter-Checks subroutine called"
1299 RETURN
1300 '
1301 ' Verify-Checks
1302 '
1310 PRINT "Verify-Checks subroutine called"
1599 RETURN
1600 '
1601 ' Enter-Deposits
1602 '
1610 PRINT "Enter-Deposits subroutine called"
1899 RETURN
1900 '
1901 ' Verify-Deposits
1902 '
1910 PRINT "Verify-Deposits subroutine called"
2199 RETURN
2200 '
2201 ' Enter-Balances
2202 '
2210 PRINT "Enter-Balances subroutine called"
2499 RETURN
2500 '
2501 ' Verify-Balances
2502 '
2510 PRINT "Verify-Balances subroutine called"
2799 RETURN
2800 '
2801 ' Calculate
2802 '
2810 PRINT "Calculate subroutine called"
3099 RETURN
3100 '
3101 ' Report
3102 '
3110 PRINT "Report subroutine called"
3399 RETURN
```

FIGURE 5.4-5: An example of test stubs that can be used to test a higher-level module. In this case, the module to be tested is the main module given in Figure 5.4-4.

as output parameters. Unfortunately, other languages do not have such facilities and must use files instead. This adds another level of complexity to the testing technique.

The main objectives in performing a top-down test are to verify the interface between the main module and the sub-module, and to test for functions that the main module performs directly, rather than by calling subroutines.

```
100 PRINT
110 PRINT "This program calculates the average of a list"
120 PRINT "of numbers entered by the user."
130 PRINT
140 GOSUB 310                       'Enter-Values
150 AVG = TOTAL / N                 'calculate the average
160 PRINT
170 PRINT "*******************************************"
180 PRINT
190 PRINT "The average is "; AVG
200 PRINT
210 INPUT "Want to average another list (Y/N)? ",ANSWER$
220 IF (ANSWER$ = "Y") OR (ANSWER$ = "y") GOTO 130
230 PRINT
240 PRINT "***** End of Program *****"
250 GOTO 999
300 '
301 ' Enter-Values
302 '
310 N = 5                     'Number of values entered
320 TOTAL = 456               'Total of all values entered
399 RETURN
999 END
```

FIGURE 5.4-6: An example of a test stub that supplies values to output parameters. In this case, the main program cannot be tested without having values for *N*, the number of values entered, and *TOTAL*, the sum of the values entered. However, note that the "data" supplied does not vary, even if the tester responds "Y" when asked if he or she would like to average another list of values. This means that the main routine is not as thoroughly tested as it could be if the lower-level module were already implemented. However, such testing is still worthwhile to eliminate the most obvious mistakes.

Bottom-Up Testing

In this case, the lowest level modules are tested first, just as they are implemented first in a bottom-up implementation method. Bottom-up testing usually follows a bottom-up implementation, but this is not in any way a requirement. Again, if the modules are being tested immediately upon implementation, this is the most natural mechanism.

With the bottom-up technique, a completed module is tested. This means that it is unnecessary to create stubs, since this module does not call any other modules. However, since this module is probably a subroutine, it is necessary to create a **test driver** program. This driver will effect a call of the subroutine, passing test data as parameters, and receiving output data in return. The driver will then probably print out the returned data for verification by the test programmer.

As in the case of test stubs, it would be ideal if the test driver could vary the values sent as input parameters to the module being tested. The READ and DATA statements could again be used in BASIC. Changing the test data for a test driver is not as much of a chore as in the case of using stubs, since there is only one test driver routine. When testing in a top-down fashion, it may be necessary to construct numerous test stubs. Consider the case of the checkbook program again (Figure 5.4-5).

Figure 5.4-7 gives an example of a typical test driver. This routine sets up values for the input parameters for the *Total-Checks* module of the checkbook program. The output parameters from the module are simply output for inspection by the testing programmer.

Top-Down vs. Bottom-Up

As is the case during the development phase, there are trade-offs between using a top-down and a bottom-up method. As mentioned above, using one of these methods for testing usually follows having used the same method for the development of the program. However, using a mixed mode of both top-down and bottom-up can be followed in testing as well as in the design phase.

When the trade-offs are taken into account, neither method appears overwhelmingly superior to the other. As is typical, what is an advantage for one is essentially a disadvantage for the other.

There are two main advantages to the top-down method. First, having an early skeleton for the program, as when most of the higher-level modules have been coded and the lower-level modules are represented by stubs, can allow some demonstration of the system. This might be helpful for training purposes, or for presentations to management. Second, once the major I/O functions have been added to the program, setting up test cases is usually easier. This results from having well-defined inputs and outputs for the program. However, this assumes that these I/O functions have not been delegated to low-level modules. Such delegation actually makes test cases harder to represent.

There are also several other disadvantages to top-down testing. First, stub modules can become more difficult to develop than originally anticipated. This is usually the result of the difficulty of varying the test data that might originate in one or more of the stubs. In

```
10 ' This is a test driver for the Total-Checks module
20 '
100 DIM CHECK(100)
108 ' set up initial test data into input parms of module
109 '
110 CHECK(1) = 125.1
120 CHECK(2) = 17.3
130 CHECK(3) = 99.99
140 NCHECKS% = 3
150 '
160 GOSUB 3510          'test the Total-Checks module
170 '
180 ' output the return parameters from the module to see if
190 ' they give the expected results.
200 '
205 PRINT
210 PRINT "TOTALCHECKS = ";TOTALCHECKS
220 PRINT
230 GOTO 9999
3500 '
3501 ' Total-Checks
3502 ' This module calculates the total of all outstanding checks
3503 '
3504 ' Input parms: NCHECKS -- the number of outstanding checks
3505 '              CHECK   -- the array that holds the checks
3506 ' Output parms: TOTALCHECKS -- sum of all outstanding checks
3507 '
3510 NCHECKS35% = NCHECKS%             'set input parameter
3520 TOTALCHECKS35 = 0                 'initialize accumulator
3530 FOR I35% = 1 TO NCHECKS35%
3540   TOTALCHECKS35 = TOTALCHECKS35 + CHECK(I35%)
3550 NEXT I35%
3560 TOTALCHECKS = TOTALCHECKS35       'set output parameter
3599 RETURN
9999 END
```

FIGURE 5.4-7: An example of a test driver routine for a bottom-up implementation and testing scheme. The "main program" here sets up some initial values for any variables that are used as input parameters for the *Total-Checks* module. This driver also ouputs values from the output parameters of the module. These outputs are used by the tester to verify that the module is working properly. More than one test run may be required to test the module thoroughly, using different sets of test data each time.

fact, because of this stub arrangement, test cases might be, at best, very difficult to devise.

Next, reviewing the results of the test might be greatly complicated if the I/O functions are not entirely completed. This makes debugging much more difficult, reducing confidence in the software.

Finally, one is easily tempted to defer testing until a lower-level module has been coded (e.g., an I/O module). This reduces the effectiveness of incremental testing and approaches the "big bang" method.

For bottom-up testing, the pluses and minuses seem equally offsetting. Test cases are usually easier to devise and implement using driver routines than with stubs because the modules being tested are small and well-defined. In addition, the test results are usually easier to observe, making the outcome of the test simpler to evaluate.

The main disadvantage of the bottom-up method is that the program itself does not exist as an entity until the last, highest level module is added. This makes demonstrating the system nearly impossible, even if ninety-five percent of the code has been written and tested.

A lesser problem is that creating the test driver itself is often more difficult than creating test stubs. However, the effort usually yields more significant results. This is because the main purpose of top-down testing is to verify the module interface, while for bottom-up testing the main purpose is to verify the internal functions of the module.

Personally, I feel more comfortable with the bottom-up testing approach. However, I'm usually forced to compromise in the same way I do with the development methods, and wind up using a combination of top-down and bottom-up testing. I attempt to use the top-down method whenever I feel that I can overcome the disadvantages discussed above. Otherwise, I use bottom-up testing.

Integration Testing

The above descriptions discuss testing modules by themselves, in isolation from other, perhaps related, modules. It is necessary at some point to test a group of modules together to verify how well they interact with one another, and that they work as a coordinated whole. This combining of modules for testing is called **integration testing.** This usually means that one module at a time is added to the program, which is then retested with the new module in place. It is not a good idea to add several new modules at one time, since this makes the test data harder to develop and makes the results more difficult to interpret.

When using the top-down technique for testing, integration means successively replacing stubs with completed modules, and redoing the tests. If a stub has been created for a higher-level module, there is a probability that this module will itself be making subroutine calls. For integration testing, it is not necessary to create stubs for these lower-level modules until the higher-level module itself has been coded and is being tested.

For instance, if the checkbook program specified in Figure 5.4-3 were being tested top-down, the first module to be tested would be *Checkbook*, the main program. This would necessitate creating stubs for the sub-modules *Enter-Checks*, *Verify-Checks*, *Enter-Deposits*, *Verify-Deposits*, *Enter-Balances*, *Verify-Balances*, *Calculate*, and *Report*. However, the stub for *Calculate* would not yet need stubs for the sub-sub-modules of *Total-Checks*, *Total-Deposits*, and *New-Balance*. When the *Calculate* stub is actually coded, it would become necessary for stubs to be created for the sub-sub-modules *Total-Checks*, *Total-Deposits*, and *New-Balance*, so that they could be called by *Calculate* during testing of that module.

For bottom-up integration, higher-level driver routines, encompassing successively more modules, must be devised. This is initially more work than replacing stubs in the top-down approach, since the driver routines change significantly from level to level. Integration begins at the level just above the lowest, following the individual tests of the lowest level modules. It continues upward until the highest level, the main module, is added. At this point the entire program is being tested as a whole.

In the checkbook program, bottom-up testing begins with the lowest level, modules *Add-Checks*, *Delete-Checks*, etc. When these have all been implemented and tested individually, testing moves up to the next level, e.g., to *Verify-Checks*. It is not necessary to test (or implement, for that matter) *all* of the modules at the lowest level before integration can begin. The main concern is that all modules *at the same level within one branch of the design* are implemented and tested together.

For instance, referring to the H-chart in Figure 5.4-3 again, the modules *Total-Checks*, *Total-Deposits*, and *New-Balance* can be tested independently of modules *Add-Checks*, *Delete-Checks*, and *Change-Checks* until the main program module, *Checkbook*, is added. In this case, we could implement and test the three modules under *Calculate* individually, then combine them in an integration test by developing a combined test driver. Next, the *Calculate* module might be implemented and tested, using the three modules under it that were already tested. This can be done prior to, for instance, implementing and testing the *Add-Checks*, *Delete-Checks*, and *Change-Checks* modules.

I typically continue following a branch upward until I can't proceed any higher because other modules on the same level have not yet been implemented. Then I return to a lower level and begin again.

Regression Testing

One final complication in incremental testing concerns the need to retest modules that have been integrated. As was seen in a previous section, each module is usually subjected to multiple tests, each one looking for a particular type of error. The main question is, How thoroughly does a lower-level module need to be tested once it is integrated, given that it has already passed its rigorous individual tests?

For example, when *Calculate* is finally coded and ready to test, do we need to set up test data (using *Calculate*'s test driver) so that *Total-Checks, Total-Deposits,* and *New-Balance* are completely retested? The justification for this would be that, even though these modules might produce the expected results when tested alone, they could produce entirely different results when tested altogether. This is especially a concern when the modules have been coded and tested by different programmers, as happens in many companies that use the team approach to software development.

This is also a concern whenever there has been a change to a module. Let's say that we are currently testing *Calculate* and a bug shows up. First, imagine that the bug has been isolated to the *Total-Deposits* module and has been fixed. The change, theoretically, is isolated to the one module. But what if, for instance, the change resulted in the use of a particular global variable that is also used by the *New-Balance* module? Without retesting *New-Balance,* this new bug might not be discovered.

Second, imagine that the original bug occurred in *Calculate.* A change to *Calculate* could affect any of the lower-level modules beneath it. For instance, the modification might have changed the way that *Calculate* calls one of the lower modules. This might require changing the lower module to match this new calling procedure. Again, retesting the lower modules might be the only way to reveal the new bug.

However, such retesting every time a module has changed is enormously time-consuming. Indeed, while necessary in some respects, full retesting of modules should not be done as a rule. Instead, tests are selectively reapplied during integration or as a result of debugging. Since the internal logic of other modules that must be retested has not changed, certain of the tests discussed previously can be ignored. It is usually sufficient to reapply the normal test case. However, where significant changes have been made to a module, including one or two of the other tests during regression testing might provide additional confidence in the program.

System Testing

An individual program, like a module, may be only one part in a large collection of programs that make up a system. Therefore, programs are often treated as modules, and systems are treated as programs, where testing is concerned.

First, each program in the system is tested by itself, as described above. Then, programs are combined for further testing. This combination is not necessarily a strict integration as previously described, since the programs do not necessarily follow as neat a hierarchy as modules within a program do. Integration of programs does not need to follow either a top-down or bottom-up order, unless there is a defined top or bottom to the relationship of the programs within the system.

As discussed in Chapter 1, programs do not often communicate directly with one another, but communicate instead through data files. Therefore, integration testing usually means using the output file from one program in the system as the input file of another. Each program would have its own test data set up in appropriate input files, or ready for inputting manually. In this way, the interfaces between the programs, i.e., the record formats of files used in common, are tested. This is the main concern when integrating programs into a system.

Occasionally, a new program is added to an existing system, or an old program is modified due to the addition of a new feature or the correction of a bug. When this occurs, it is necessary to perform some level of regression testing on the system as a whole, in addition to the modules within the program being modified or written. It is necessary to verify that new code in the system does not produce a bug elsewhere within the system.

5·5 DEBUGGING TECHNIQUES

In this section we are going to explore many of the techniques commonly used by professional programmers to debug their programs. The previous sections dealt with identifying that bugs exist. This section's goals are first to locate those bugs precisely, and then to remove them from the program.

Locating the bug is not as simple as it might at first appear. As we've discussed several times, how a bug manifests itself does not always give adequate clues to its whereabouts in the program. Locating the bug is usually the more difficult of the two goals, and consequently takes the most time. Indeed, once the source of the bug is known, fixing it is usually fairly straightforward. Recall that one of the arguments given for using the various methods discussed throughout this book is that such techniques make fixing bugs much easier. This is discussed in more detail in Chapter 7, Software Maintenance.

There are several reasons why locating a bug is so difficult. To begin with, there are a number of factors that may have caused the bug. It could be a result of missing code, extra code, or code improperly stated. There is no way to tell from the tests which of these situations applies.

A more devious problem is that there may, in fact, be more than one thing wrong with the program. This compounds the difficulty of debugging in the same way that a patient having multiple illnesses complicates a diagnosis for a physician. The combination of symptoms may make the physician think that the patient has only one disease, when in fact he or she has more than one.

I have dubbed this problem the Gopher Principle #1, which states, "Where there is evidence of a gopher in the yard, there is undoubtedly more than one." Any attempt to eliminate only one gopher will probably not be successful; one has to be prepared to deal with a whole family of them. The only reasonable approach is to tackle one bug source in your program at a time, until you are certain that all contributing factors have been accounted for.

However, the program you are debugging might not be one you wrote yourself. For professional programmers, this is an everyday occurrence. There are two main difficulties to overcome in this situation. First, the original programmer may not have adhered to the methods of structured design or implementation that you would have followed. This makes adding or changing code more difficult. Second, the code is probably unfamiliar to you, even if you have looked at it before. This can be particularly troublesome if the programmer did not take care in developing adequate documentation for the system. It is nearly impossible to understand a program simply by reading the code. Unfortunately, this is often all you have to go on.

The debugging task takes place at two different points in the life cycle of a program. First, it is done in conjunction with the testing phase to locate and eliminate implementation bugs. The second time it is done is after the program has entered **production phase,** i.e., when it is being used for real work by end users. In this case, the programmer is notified by a user that something has gone wrong. While most errors are traceable to user error, it is still usually the responsibility of the programmer to identify the error conclusively, and to remedy the situation. The remedy often means changing the program to protect against that type of error in the future.

Getting the Bugs Out

Recall from the previous section that one of the main results hoped for in designing test cases is to smoke out bugs. This testing attempts to force errors to show up in four main categories: errors in the mechanical coding of the program (such as a syntax error), errors in the logic of the program, errors in the data (e.g., out of range), and errors in the definition of the program (i.e., it does not meet the specifications of the user).

When a bug is first discovered, it is not usually obvious exactly what the source of the problem is. An error message generated either by the operating system or by the program itself is not often very helpful in locating the source of the error, or even in locating what line of code produced the error. To make matters worse, the particular error that occurred may be intermittent in nature. Debugging a program is a lot like fixing a car, in this respect. There is nothing more frustrating than taking your car to a mechanic, only to have the problem go into hiding as long as the mechanic is listening.

A paraphrase of the Heisenberg Uncertainty Principle says "Things change under observation." In other words, merely looking at something can change the way it behaves. This seems to be especially true when the observer is a professional whose job it is to excise the bug.

Part of the task in debugging a program, then, is to make the program "fail reliably." This means that the programmer must be able to reproduce the error under controlled testing procedures. This is not always a trivial matter, and in some cases it is impossible on a practical level.

The first step is to collect as much information from the user as possible about the manifestation of the error. (If the error occurred during initial testing of the program, this information is already available.) The main information of interest is a description of the inputs and outputs involved. This may be somewhat difficult to determine if you're dealing with a complex, highly interactive system. But it is necessary to trace what lines of code have been executed, and in what order, and having all the inputs and outputs makes this much simpler. In addition, it is possible to locate the source of the error by using the generated outputs to trace the path of execution through the code.

Second, the program should be retested using new test cases in an attempt to force the error to occur again. Using the original test data will do little good. If the bug didn't show up before when this data was used, it won't show up in retesting, since the code should execute in exactly the same way every time, given the exact same inputs.

Creating new test cases requires the formation of a hypothesis about the nature of the bug. The test cases subject the program to an experiment which is used either to confirm or disprove the hypothesis. Unsuccessful tests require reformulation of the hypothesis and retesting, until a version of the hypothesis is confirmed. This subjects the program to what is commonly called the **scientific method** using **inductive reasoning.**

The next step is to attempt to correct the problem by "patching" the program code. This could mean something as simple as deleting one line of code, changing a + sign to a − sign, or adding a few lines of code. On the other hand, the "fix" might require adding new modules to the program, which would mean going through most of the steps of design and development that have already been discussed. It is never obvious in the beginning just how extensive any changes will need to be to correct a bug.

There are three temptations that must be avoided at all cost when facing a debugging task. The first temptation is to make changes to the program in a trial-and-error attempt at correcting the problem. In this case, avoid jumping to conclusions by rushing from the formulation of a hypothesis to patching the code. Take time to test and refine the hypothesis thoroughly before changing any code.

Next, it is far too tempting to treat just the symptoms of the bug, instead of attacking the bug itself. If, for instance, you notice that the program consistently gives the value of a certain output as being one greater than it is supposed to be, don't simply put a statement subtracting one from that value before it is output. Find out exactly why this value is larger than it should be, and correct that.

Finally, it is tempting to begin coding a fix as soon as you have any idea what the source of the problem is. Even if you are right, just adding code could destroy the careful work you did when you originally implemented the program. As discussed in Chapter 7, Software Maintenance, any code added to a program should follow exactly the same format and design philosophy as the original code. Otherwise, after just a couple of patches are added to a program, you won't be able to tell that you used structured techniques at all!

The final step in debugging is to retest the program. This should be done with as much care as the original testing. First, you must verify that the changes made did indeed correct the noted error. This can usually be done by using the same test cases devised above to duplicate the error. In addition, however, it is important that the program be retested to some extent using the original test cases. This follows the philosophy of regression testing. The object is to make certain that any changes that have been made have not introduced new bugs into the program. While regression testing does not always generate this, it will at least eliminate the most obvious new bugs.

To summarize, the following steps should be followed when debugging a program:

1. Verify that an error exists.
   - **a.** collect pertinent information from users
   - **b.** prepare new test cases to duplicate the error
2. Locate the error
   - **a.** form a hypothesis about the error's source
   - **b.** use tests, outputs, and other debugging aids to verify the hypothesis
   - **c.** refine the hypothesis until the error is located
3. Fix the error
   - **a.** add, delete, or modify code as required
   - **b.** retain original structure of code
4. Retest the program
   - **a.** use new tests to verify that the fix was effective
   - **b.** use original tests to verify that changes did not introduce new bugs

Built-in Aids

The amount of help that a programmer gets from any language system in debugging a program varies greatly from language to language. Some language systems help very little. This is especially true of most interpreted languages, such as most versions of BASIC, since they are much more limited in what they can do with the source code of a program than a compiled language. Other languages take several alternative approaches to supporting the debugging effort.

The approaches are of two types, **static** and **dynamic.** The static aids come in the form of listings of the code in various formats. These listings help collect and organize certain information about the code that the programmer often finds useful during debugging.

Dynamic tools are usually software separate from the compiler or interpreter itself. These **debuggers** will be discussed in the next section. The most often used built-in dynamic tool is a tracing facility. This traces which lines of code were executed before an error occurred.

General Listings

For interpreted languages there is usually only one type of listing of the program. This gives only the source code itself. While such a listing is ultimately necessary when debugging, it does not generally enhance the programmer's ability to do the debugging. This is a rather important limitation to the use of interpreters. Therefore, the following discussion will mainly concern compiler listings.

The format of the main listing generated by a compiler differs both from language to language and from compiler to compiler for the same language. Figures 5.5-1, 5.5-3, and 5.5-4 show compiler listings for a compiled version of BASIC and for two versions of Pascal. While all of the compilers list the source code, each includes slightly different additional information in the listing.

Figure 5.5-1 shows a compiler listing for the IBM BASIC compiler, which is a version of the Microsoft BASIC compiler. It is only a report of one phase of the compilation and is not directly executable. There is little additional information provided by this listing. The first column provides an **offset** for each line of code. This is simply a length, in hexadecimal form, from the beginning of the program to the beginning of the line of code. The second column is the length of memory currently needed to hold all the variables used up to this point in the code.

These values can be used to aid debugging in only the most painful ways. It turns out that if the program crashes, it will generate an error message, in addition to some indicator of where (i.e., what line of code) in the program the bug occurred. Unfortunately, this indicator is often given as an offset in hexadecimal, rather than as a line number. It is then necessary to use another listing, called a

```
                                          IBM Personal Computer
Offset      Data        Source Line       BASIC Compiler V1.00
 001A       0002        100 GOSUB 210
 002E       0002        110 GOTO 999
 0031       0002        210 FOR I = 1 TO 10
 0039       0002        220   FOR J = 21 TO 25
 0041       0002        230     PRINT I, J
 004D       000A        240     GOSUB 510
 0050       000A        250   NEXT J
 0066       000A        260 NEXT I
 007C       000A        299 RETURN
 007D       000A        510 K = I * J
 008A       000E        520 PRINT K
 0091       000E        599 RETURN
 0092       000E        999 END
 0095       000E
 0098       000F

22151 Bytes Available
21828 Bytes Free

    0 Warning Error(s)
    0 Severe Error(s)
```

FIGURE 5.5-1: A listing generated by the IBM BASIC Compiler.

map (see Figure 5.5-2), generated by the language's **linker** (software that prepares a program for execution following compilation) and another special program called a **debugger** in order to discover the real location where the error occurred. As mentioned above, this is an extremely painful mechanism to use, since it deals with several listings, additional software, and (egad!) hexadecimal arithmetic. Fortunately, this can all usually be avoided by other methods discussed later in this chapter. In addition, the IBM compiler provides a switch that can be "thrown" during compilation which forces execution error messages to include the line number of the error instead of an offset.

The third column in Figure 5.5-1 provides the BASIC source code listing, in exactly the same format as it would be given if you asked for a LIST when using the BASIC interpreter. Again, however, there is no additional information that might be helpful when debugging.

This compiler does provide some capabilities for formatting the listing in order to make it more readable, however. This is done by using what IBM calls **metacommands,** which are also sometimes called **pseudo-instructions.** These commands are included directly in the

```
 Start     Stop      Length    Name        Class
 00000H    00097H    0098H     BC_CODE     CODE
 00098H    000F2H    005BH     CODE        CODE
 00100H    0010FH    0010H     BC_ICN      INIT
 00110H    00112H    0003H     BC_IDS      INIT
 00120H    0031CH    01FDH     INIT        INIT
 00320H    00F5FH    0C3FH     CONST       RT_DATA
 00F60H    00F60H    0000H     DATA        RT_DATA
 00F60H    00F60H    0000H     COMMON      BLANK
 00F60H    00F60H    000H      CONST       CONST
 00F60H    00F60H    0000H     DATA        DATA
 00F60H    00F6DH    000EH     BC_DATA     DATA
 00F6EH    00F6FH    0000H     BC_FT       DATA
 00F70H    00F7FH    0010H     BC_CN       DATA
 00F80H    00F82H    0003H     BC_DS       DATA
 00F90H    0118FH    0200H     STACK       STACK
 Program entry point at 0000:001A
```

FIGURE 5.5-2: A map generated by the IBM BASIC Compiler's linker. This map can be used as an aid in debugging but is difficult at best to understand and use properly.

code as part of REM statements. At compile time, the compiler uses these commands to determine how it is supposed to do things.

An example is the $PAGE metacommand which forces the compiler to begin a new page before listing any more code. In this way, the listing can be broken up into modules, making it somewhat easier to locate and read specific sections of the code. Another command, $TITLE, writes a title at the top of the page. This, again, lets the programmer create a listing that is somewhat more readable than usual.

The IBM Pascal Compiler produces a different type of listing than that seen with the BASIC compiler. Figure 5.5-3 shows an example. In addition to the source code listing and the added line numbers (Pascal statements are not numbered), there are four columns that provide us with useful information.

The "J" column contains flags which describe various kinds of jumps that have been included in the code. A statement that contains a forward jump (e.g., a GOTO) is flagged with a plus(+) sign in this column. A backward jump is flagged with a minus(−) sign. All other jumps, such as a RETURN statement in a procedure (i.e., subroutine) are flagged with an asterisk (*). This information is useful in that it helps to identify all jumps within the code. It has been shown in research that jumps are the most frequent cause of bugs in a pro-

```
JG IC   Line#  Source Line          IBM Personal Computer Pascal Compiler V1.00
   20      1   program nonsense(input,output);
   10      2   var global: integer;
           3
   20      4   procedure proc2(l,m:integer; var n:integer);
   20      5   begin
   21      6     n := l * m;
 = 21      7     global := 42;
   21      8     writeln(n)
   10      9   end;
Symtab     9     Offset  Length  Variable - PROC2
               -    6      10    Return offset, Frame length
               -    0       2    L                                    :Integer ValueP
               -    2       2    M                                    :Integer ValueP
               -    4       2    N                                    :Integer VarP
          10
   10     11   procedure procl;
   20     12   var i, j, k: integer;
   20     13   begin
   21     14     for i := 1 to 10 do
   21     15       for j := 21 to 25 do
   21     16       begin
   22     17         writeln(i,j);
 = 22     18         global := 23;
   22     19         proc2(i,j,k)
   22     20       end
   10     21   end;

Symtab    21     Offset  Length  Variable - PROC1
               -    0      10    Return offset, Frame length
               -    4       2    I                                          :Integer
               -    6       2    J                                          :Integer
               -    8       2    K                                          :Integer
          22
   10     23   begin              {main program}
   11     24     procl
   00     25   end.
Symtab    25     Offset  Length  Variable
                    0       4    Return offset, Frame length
                    2       2    GLOBAL                               :Integer Static
                 Errors   Warns  In Pass One
                    0       0
                    ^       ^

NONSENSE
```

FIGURE 5.5-3: An example of a listing generated by the IBM Pascal Compiler. This listing offers many useful pieces of information for debugging purposes.

gram. With these statements flagged, we can verify by hand that the jumps indeed do what we intended.

The second column is the "G," or global, column. Here all global variables are flagged. This is especially helpful in spotting dangerous situations, such as side effects, caused by global variables. Remember that side effects occur as a result of an inadvertent change to a variable from within a subroutine to which the variable is global. The flag in this column can also point out dangerous misuses of parameters in procedure calls.

The third column, labeled "I," contains the **level** of all identifiers (names of variables, procedures, types, etc.). This level indicator changes whenever a procedure, function, or record declaration is encountered. It increases when the declaration is first found, and decreases when the **scope** of the procedure (the code that is contained within the procedure) ends.

The "C" column shows the level of control statements. It changes with each BEGIN/END block, CASE/END statement, and REPEAT/UNTIL loop.

The "I" and "C" columns are used to help determine that all the code is arranged in blocks properly. It can, for instance, tell you if you are missing a BEGIN/END pair in a block of code. By identifying the scope of each block (both control statements and procedures), these columns can be used to verify that the computer groups statements the way you think it should. If it doesn't (i.e., if the scopes of the blocks are not what you think they should be), then you have not properly specified one or more blocks of code.

This compiler listing also includes something called a **symbol table.** This is a listing of all the variables used within each procedure, function, and program. In addition to specifying the offset within the block and the length (size) of each variable, the table lists the type of each variable (e.g., integer, floating point, etc.). It also gives useful information about any parameters that are being passed to/from procedures or functions. This information should be used to verify that each variable is of the proper type.

This compiler has metacommands similar in function to those available with the BASIC compiler discussed above. They can be used to set up headings on each page, to partition the code into manageable groupings, and to change the way the compiler checks for errors. Only a few of these features help in the debugging effort in any direct way, but they all make it somewhat simpler.

As a final example, consider the listing in Figure 5.5-4. This listing is generated by the UCSD Pascal p-system, which is available on a variety of computers, in this case a DEC Rainbow 100. The leftmost column is simply line numbers. The second column gives what the UCSD p-system calls the **segment** number. Most programs are composed of a single segment, and so the numbers are all the same. If additional code had been included in the program from

```
Pascal Compiler IV.1 c5s-4      8/23/84


    1   2    1:d    1  program nonsense(input,output);
    2   2    1:d    1  var global: integer;
    3   2    1:d    2
    4   2    1:d    2  procedure proc2(l,m:integer; var n:integer);
    5   2    2:0    0  begin
    6   2    2:1    0    n := l * m;
    7   2    2:1    5    global := 42;
    8   2    2:1    9    writeln(n)
    9   2    1:0    0  end;
   10   2    1:0    0
   11   2    1:d    1  procedure proc1;
   12   2    3:d    1  var i, j, k: integer;
   13   2    3:0    0  begin
   14   2    3:1    0    for i := 1 to 10 do
   15   2    3:2    9      for j := 21 to 25 do
   16   2    3:3   18      begin
   17   2    3:4   18        writeln(i,j);
   18   2    3:4   43        global := 23;
   19   2    3:4   46        proc2(i,j,k)
   20   2    3:3   49      end
   21   2    1:0    0   end;
   22   2    1:0    0
   23   2    1:0    0   begin          { main program }
   24   2    1:1    0     proc1
   25   2     :0    0   end.

End of Compilation.
```

FIGURE 5.5-4: An example of the compiler listing generated by the UCSD Pascal compiler. This listing does not offer as much detail as IBM's Pascal compiler.

another program file, this additional code would have a different segment number.

The third column is in two parts and is very similar to the "I" and "C" columns of the IBM Pascal compiler. The first part is the **routine number,** which links each line of code with the module to which it belongs. The second part is called the **lexical level,** which identifies levels of code scope within each module. This number changes with each BEGIN/END block and each new control structure, and shows a "d" for all declarations.

Error Messages

Compiler listings also include messages about any errors that were encountered during the compilation. The complexity and usefulness of the messages generated by each compiler differ as much as the general listings do. Typically, a compiler can only detect syntax errors. The listing in Figure 5.5-5 shows an example using UCSD Pascal. In this case, the actual error was that the procedure *proc2* was called (from *proc1*) before it was officially declared. This resulted in the compiler being confused with the *proc2* reference in line 10. It tried to interpret the reference as a variable, and so generated the error message "Undeclared identifier." Since it thought that this was a reference to a variable instead of to a procedure, it also looked for an assignment statement. This generated the "' := ' expected" message.

As is obvious from this example, even though the compiler knows

FIGURE 5.5-5: An example of how the UCSD Pascal system detects syntax errors. Note that there is some amount of confusion on the part of the compiler about exactly what the error is. This is a fairly common problem for compilers, since they typically possess only a modest amount of "intelligence."

```
Pascal Compiler IV.1 c5s-4      8/23/84


    1   2    1:d     1   program nonsense(input,output);
    2   2    1:d     1   var global: integer;
    3   2    1:d     2   procedure proc1;
    4   2    2:d     1   var i, j, k: integer;
    5   2    2:0     0   begin
    6   2    2:1     0     for i := 1 to 10 do
    7   2    2:2     7       for j := 21 to 25 do
    8   2    2:3    14       begin
    9   2    2:4    14         writeln(i,j);
Syntax Error: Undeclared identifier
Syntax Error: Error in variable
      Previous error - line   10
   10   2    2:4    39         proc2(i,j,k)
Syntax Error: ':=' expected
      Previous error - line   10
   11   2    2:3    39       end
   12   2    1:0     0    end;
   13   2    1:d     1    procedure proc2(l,m:integer; var n:integer);
   14   2    3:0     0    begin
   15   2    3:1     0      n := l * m;
   16   2    3:1     5      writeln(n)
   17   2    1:0     0    end;
   18   2    1:0     0    begin
   19   2    1:1     0      proc1
   20   2     :0     0    end.

Last syntax error: line  11
End of Compilation.
```

something is wrong, it doesn't necessarily know *what* is wrong. As a result, it generates what are apparently incorrect error messages. Unless you know how compilers work, it isn't possible to explain just why the messages are "wrong." The lesson here is that it is all too possible for a compiler to get confused. In addition, it occasionally happens that the compiler gets so confused that it goes into an apparent fit, generating multiple error messages for every line of code. This can occur, for instance, when quotation marks are used improperly.

Many compilers generate error messages on multiple levels. This is especially true for compilers on large-scale systems, where error messages can have, for example, six levels of severity. Microcomputer compilers seem to limit their error messages to at most two levels. The first level is often called a **warning message.** The type of error that generates this message does not stop the compiler from translating the line of code into machine language, but wants the programmer to be aware that he or she has perhaps done something that *might* cause a problem.

For example, Figure 5.5-6 is a listing generated by the IBM Pascal compiler. In this case, there are two problems. First, the keyword "DO" was left off of the two FOR statements (lines 4 and 6). The second problem (line 11) is that a comma was typed instead of a semicolon. While, theoretically, these errors won't affect the execution of the program, the compiler just wants you to know that something isn't quite right. The program would still be executable, however, as long as the messages generated by the compiler were only warnings.

The second level of error usually included in compiler listings is the **fatal,** or **severe,** error. A program with such a compiler error cannot be executed. In fact, most compilers won't even generate the object code file if it encounters one of these errors during compilation. The program in Figure 5.5-5 has fatal errors.

It occasionally occurs that the compiler hits such a terrible error that it cannot even continue the compilation. These are often referred to as **unrecoverable** errors, because the compiler can't figure out how to continue compiling. Leaving the "program" keyword off the first line of a Pascal program will often generate such an error.

A complete understanding of what error messages really mean, and how they might be generated, is important to the programmer. Don't be misled by thinking that the error message is entirely accurate. It isn't too hard to confuse a compiler. Study every message

```
JG IC   Line#  Source Line           IBM Personal Computer Pascal Compiler V1.00
   20       1  program dummy(input, output);
   10       2  var i, j, k: integer;
   10       3  begin
   11       4    for i := 1 to 10
   11^      5    begin
            4  ---------------^Warning 172 Insert DO
   12       6      for j := 1 to 15
   12^      7      begin
            6  -----------------^Warning 172 Insert DO
   13       8        writeln(i);
   13       9        writeln(j)
   12      10      end;
   12      11      k := i * j,
           11  -------------^Warning 156, Assumed ;
   12      12      writeln(k)
   12      13    end
   00      14  end.
Symtab     14   Offset  Length   Variable
                     0       8   Return offset, Frame length
                     2       2   I                                        :Integer Static
                     4       2   J                                        :Integer Static
                     6       2   K                                        :Integer Static
                Errors   Warns  In Pass One
                     0       3
```

FIGURE 5.5-6: Another example using the IBM Pascal Compiler. In this case, warning messages are given to the user to identify things that are incorrect, but that the compiler can temporarily fix. Such nonfatal errors should be completely eliminated from the program, even though the program might still execute *properly*.

carefully for every line that gets flagged, even if you suspect that it is inaccurate or the result of a confused compiler.

It is a good idea to keep a log of the types of errors that you encounter. This is especially handy when the system documentation is a little on the sketchy side, or when certain combinations of code consistently confuse the compiler. Most important is keeping track of any misleading errors, such as when the compiler gets confused. Write down the messages that are generated, give an example of the code that generated the messages, and list the fix applied to correct the error. Compiler listings should be included whenever possible.

This log will keep you from spending long hours trying to figure out obscure errors more than once. In addition, it will make you more familiar with the way the compiler works, making you a more knowledgeable programmer.

Special Listings

Many programming systems provide other types of tools to produce special listings of programs. Mentioned in section 3.3, Program Style, was a program called a "pretty printer," which formats a program listing with indentation of blocks. This makes the listing easier to read, since it automatically identifies the scope of each block of code. IBM provides this function in a package called the "BASIC Programming Development System." Figure 5.5-7 shows *before* (a) and *after* (b) listings of a program that was run through the formatter.

Another part of this package is a program that produces a cross-reference listing of all the variables and keywords used throughout the program. As shown in Figure 5.5-8, this listing gives statement numbers that have been referenced in the program (e.g., with a GOTO), and the lines on which these references occurred. This is helpful in spotting GOTOs to nonexistent or incorrect line numbers, and for finding misspelled variable names, locating keywords used as variable names, and places where the wrong variable names were used.

Tracing

It is not usually very obvious, even after studying any output produced by a program during its execution, what lines of code were actually executed before an error halted the action. It is important, however, to have an idea of what lines were executed, in what order, and how many times. Without this information, debugging becomes guesswork, which is only one notch above voodoo.

One of the nice features of an interpreted language is that, since a line of source code must be translated into machine language *every* time it is executed, it is possible to retain more specific information about the source code during execution. In a compiler system, the source code is nowhere in sight during the actual execution of the program.

For BASIC, this means that the actual line numbers given to each line of code are still available. (This is not true for compiled BASIC, however.) This being the case, we could have the interpreter point out the line number of every line that gets executed, in the order of execution. The TRON (TRace ON) command sets this up. Issuing TRON prior to running the program makes the interpreter list every line number that it executes.

Figure 5.5-9 shows an example of such a trace. The numbers in

```
100  'This is an example of the IBM BASIC formatter
110  DIM A(10)
120  PRINT: PRINT: PRINT "Hello": PRINT
130  'These lines should get indented
140  FOR I = 1 TO 10
150  FOR J = 21 TO 25
160  PRINT I, J
170  NEXT J
180  PRINT
190  NEXT I
200  IF Q = 42 THEN PRINT "NO" ELSE PRINT "YES"
210  PRINT "done"
999  END
```

a

```
   IBM Personal Computer BASIC Formatter and Cross-Reference V 1.00
100  'This is an example of the IBM BASIC formatter
110  DIM
       A(10)
120  PRINT:
     PRINT:
     PRINT "Hello":
     PRINT
130  'These lines should get indented
140  FOR I = 1 TO 10
150  |  FOR J = 21 TO 25
160  |  | PRINT I, J
170  |  NEXT J
180  |  PRINT
190  NEXT I
200  IF
       Q = 42
         THEN
           PRINT "NO"
         ELSE
           PRINT "YES"
210  PRINT "done"
999  END
```

b

FIGURE 5.5-7a and b: An example of using the IBM BASIC Development System. The listing in (a) shows how the program appears before using the BASIC formatter. The listing in (b) shows the result of the formatting. The result may not necessarily be exactly what one would like, but it is definitely more readable than the original.

the square ([ ]) brackets are the line numbers. Note that the line numbers are printed out interspersed with the output from the program. This is something of a disadvantage, as it usually messes up whatever output formats you may have devised. However, the usual procedure is to make two runs, the first without the trace (it can be

```
IBM Personal Computer BASIC Formatter and Cross-Reference V 1.00

Variable   References
A             110
DIM           110
ELSE          200
END           999
FOR           140     150
I             140     160     190
IF            200
J             150     160     170
NEXT          170     190
PRINT         120     120     120     120     160     180     200
              200     210
Q             200
THEN          200
TO            140     150
```

FIGURE 5.5-8: A cross-reference table for all variables and keywords used in the program.

turned off using TROFF), the second with the trace. This gives, in addition to the traced listing, a good clean copy of just the output from the program, which is also helpful for debugging.

Use the trace along with hand execution to determine if the program executed the statements in the order that you would expect. If the trace turns up an anomaly, you can usually follow it to the source of the error. For instance, if a loop is executed the wrong number of times, the trace will show either too many or too few executions of the loop through either too many or too few groups of statement numbers that make up the loop.

Debuggers

A debugger is a software tool that allows the programmer to have a "window" into the actual code of a program that is being executed. This code is usually machine language statements, created during

```
[110][120]
[130][310]How many?3
[315][320][330]Enter number: 17
[335][340][350][330]Enter number: 42
[335][340][350][330]Enter number: 65
[335][340][350][399][140]
[145][410][420] 17 [430][420] 42 [430][420] 65 [430][440]
[499][150][120]
[130][310]How many?2
[315][320][330]Enter number: 4
[335][340][350][330]Enter number: 27
[335][340][350][399][140]
[145][410][420] 4 [430][420] 27 [430][440]
[499][150][120]
[130][310]How many?1
[315][320][330]Enter number: 10
[335][340][350][399][140]
[145][410][420] 10 [430][440]
[499][150][160][999]
Ok
```

FIGURE 5.5-9: An example of a trace in BASIC. The numbers in square brackets indicate the line numbers that have been executed. Note that the regular output (and input) gets mixed in with the trace listing. This makes the trace rather difficult to read.

the compiling process. As a result, such code is generally not directly understandable to a human being. When looking at machine language code, all instructions, variables, etc., are generally represented only as hexadecimal numbers, rather than with names. In addition, each higher-level language statement is generally translated into several machine language instructions, making the match between the source code and the machine code difficult.

One example of such a debugger is called DDT (Dynamic Debugging Tool), and is part of the CP/M operating system. It was originally designed to aid in debugging assembly language programs rather than programs in higher-level languages such as BASIC or Pascal. It includes commands, for instance, to assemble (which is analogous to compiling for a high-level language) assembly language statements into machine language, and to disassemble machine language statements back into assembly code. This latter function is useful for locating errant branches (i.e., GOTOs) in the code, since branching in assembly code is generally done by distance rather than to a specific line of code by address.

More important is the debugger's ability to display memory in

```
F>ddt ckbook.bas
DDT VERS 2.2
NEXT  PC
0700 0100
-d200
0200 6F 75 73 20 73 74 61 74 65 6D 65 6E 74 2E 20 20 ous statement.
0210 46 69 6E 61 6C 6C 79 2C 20 79 6F 75 20 77 69 6C Finally, you wil
0220 6C 20 65 6E 74 65 72 20 74 68 65 20 6C 61 73 74 l enter the last
0230 22 00 A5 0B AA 00 91 20 22 61 6D 6F 75 6E 74 20 "...... "amount
0240 77 72 69 74 74 65 6E 20 69 6E 20 79 6F 75 72 20 written in your
0250 63 68 65 63 6B 62 6F 6F 6B 20 72 65 67 69 73 74 checkbook regist
0260 65 72 2E 20 20 54 68 65 20 70 72 6F 67 72 61 6D er.  The program
0270 22 00 DD 0B B4 00 91 20 22 77 69 6C 6C 20 74 68 "...... "will th
0280 65 6E 20 74 65 6C 6C 20 79 6F 75 20 69 66 20 79 en tell you if y
0290 6F 75 72 20 61 63 63 6F 75 6E 74 20 69 73 20 62 our account is b
02A0 61 6C 61 6E 63 65 64 2E 22 00 E3 0B BE 00 91 00 alanced.".......
02B0 E9 0B C8 00 91 00 F1 0B D2 00 3A 8F DB 00 10 0C ..........:.....
-d
02C0 DC 00 3A 8F DB 20 49 6E 69 74 69 61 6C 69 7A 61 ..:.. Initializa
02D0 74 69 6F 6E 20 73 65 63 74 69 6F 6E 00 18 0C E6 tion section....
02E0 00 3A 8F DB 00 55 0C F0 00 92 20 20 09 3A 8F DB .:...U....  .:..
02F0 73 65 74 73 20 61 6C 6C 20 6E 75 6D 65 72 69 63 sets all numeric
0300 20 76 61 6C 75 65 73 20 74 6F 20 30 2C 20 61 6C  values to 0, al
0310 6C 20 73 74 72 69 6E 67 73 20 74 6F 20 6E 75 6C l strings to nul
0320 6C 00 62 0C FA 00 BA 20 42 41 53 45 20 31 00 6A l.b.... BASE 1.j
0330 0C 04 01 3A 8F DB 00 9C 0C 0E 01 91 20 22 2A 2A ...:........ "**
0340 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A ****************
0350 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A ****************
0360 2A 2A 2A 2A 2A 2A 2A 22 00 C2 0C 18 01 46 4F 52 *******".....FOR
0370 4D 41 54 31 24 20 F0 20 22 5C 20 20 20 20 5C 20 MAT1$ . "\    \
-
```

FIGURE 5.5-10: An example of using the CP/M DDT program on a file. In this case, the file was a BASIC program. More usual is to use such a tool on the object code of a compiled or assembled program. Such a listing allows us to see the ASCII codes that are directly stored in the file.

various formats. The most useful format produced with DDT is with the D (display) command. Figure 5.5-10 shows an example of a particular file that was loaded into memory. It displays the contents at each location (the first column gives the memory address) in hexadecimal. In addition, the right-hand column translates the hexadecimal codes into ASCII characters whenever possible. Some hexadecimal codes do not translate into printable ASCII characters, and are represented instead by a period in the right-hand listing.

This technique can sometimes be used to look at memory following the execution of a program. This would allow one to examine the contents of variables when execution terminated. If the program ended with a bug, these values could be important in the effort to decipher the source of the bug.

Unfortunately, this technique is both cumbersome and error prone. It is not always possible to get a clean picture of the executed program, because the error that caused the program to halt may have wiped out portions of memory, making these sections useless to view. In addition, invoking DDT itself can destroy anything currently in memory. However, the technique of peeking at the contents of memory (i.e., variables) is quite useful during the debugging task.

Another example of a debugger is the one available with the UCSD Pascal system. With it, the programmer can set what are called **breakpoints,** places in the program where execution will halt and the debugger is invoked. This allows the programmer to view the contents of variables during the middle of a program's execution. Once viewing is completed, the execution can be resumed. In addition to being able to view the variables' values using this debugger, the programmer can change the values during the program's execution.

Setting multiple breakpoints throughout the code lets the programmer set up what are called **snapshots** of the program's execution. These snapshots are listings of the intermediate values of variables, values of variables in between any outputs normally generated by the program.

In its standard form, the method of using the UCSD debugger is again cumbersome. The debugger refers to segment numbers, offsets, and lexical levels in order to locate a variable for viewing or altering. The programmer must use the numbers generated on the compiler listing in order to do anything of interest.

However, there is an option with the UCSD debugger to perform symbolic debugging. This allows the program to refer to a procedure or variable by name instead of by number. Using symbolic debugging requires that a special format of the program be compiled. Including a "{$D+}" at the beginning of the program will cause special code to be included in the object code which allows for symbolic debugging. This makes life a lot easier.

The trade-off is that the program will execute many times slower than normal because of the extra code. This is certainly not a problem for small programs, or for programs that require a lot of interaction from the user. For larger programs where speed is more crucial, you could use the D switch during the testing phase, then remove it and recompile the program before it is put into actual use. This would remove the debug statements from the final copy of the object code.

SOP Debugging

There's a procedure in road rallying called SOP (seat of the pants) driving. This means you rely only upon pencil and paper and the car's speedometer and odometer to make all your calculations. (The object in rallying is to follow an obscure path in a fixed amount of time.) This is as opposed to using calculators and car computers, which automatically calculate, for instance, how fast you need to go over the next five miles in order to hit the next checkpoint exactly on time. SOP driving is challenging, and is sometimes more fun than computer-aided rallying, but it is obviously more difficult to do right. It can, however, yield, more rewarding results than relying on the computer to do everything for you.

SOP debugging is very similar to the SOP driving in rallying. It is somewhat more difficult because it is mostly mechanical (at least initially), and because you, as the programmer, must do things that a debugger might do for you. Unlike rallying, however, SOP debugging is not resorted to just for the fun or challenge of it, but because either the programmer doesn't have a decent debugger, or because debuggers can't be used. This is most often the case when using an interpreted language, such as BASIC.

In essence, the main objective is to emulate what a debugger might do by including extra code in the program, and by doing other activities by hand.

Hand Execution

The first weapon in the SOP debugging arsenal is the old reliable hand execution method. This was covered in detail in section 5.3, Desk Checking. This technique is slow and methodical in its search for the source of a bug, and finding the bug cannot typically be done entirely with this method. You must also rely upon the output that was generated by the program before it halted with its error.

Using a **backtracking** technique from the point of the error is often helpful. In this case you start at the last instruction that was executed and work your way backward through the code. This requires knowing the values of variables at the time execution halted. With this knowledge, you can generally deduce what previous values must have been in order for execution to have followed the path it did. You can also use other outputs generated prior to the halt to verify the path taken.

Another technique is to select some midpoint in the execution at which you assume everything was still executing properly, and trace the program forward. This is somewhat risky, in that your assumption might be wrong. However, given that the program may have been executing for a long time, perhaps consuming hundreds or thousands of inputs, this is sometimes the only reasonable approach. If you can't find the error after the selected point, you may have to go back further in the execution and start tracing forward again.

Tracing

Not all languages have the ability to produce statement traces the way interpreted BASIC does, for several reasons. First, compiled languages do not retain statement addresses in human-readable form in the object code. Second, most high-level languages do not even have distinct identifiers (such as line numbers) for each statement.

For BASIC, this makes an interpreter a useful debugging tool. This is even suggested by Microsoft in its compiler documentation. I do not entirely agree with this method, since a compiler is more efficient at discovering syntax errors than an interpreter, but for the purposes of tracing the code, it makes sense.

In other languages, I typically employ what has been called a **wolf fence** [40] method of tracing. In this technique, I select specific points in the program where I will include output statements that identify the location of the code that has just been or is about to be executed. It's referred to as a wolf fence based on the following scenario:

There is a wolf howling somewhere in Alaska. Your job is to find that wolf. One method is to construct a fence all the way across Alaska at some point (e.g., the middle) and listen for the howl again. When the wolf howls the next time, you will know from which side of the fence the howl came. Construct a similar fence on that side, and listen for the howl again. Doing this iteratively will eventually bring you face to face with the wolf.

The wolf in our situation is the bug you are trying to find. Figure 5.5-11 shows an example in Pascal. Since using line numbers is usually inaccurate for describing a location in the execution in Pascal, I usually just make up my own sequential numbers for each fence output. Useful points to include in the fence information are the beginning and end of any block or module. This would include any

```
program nonsense(input,output);
var global: integer;

procedure proc2(l,m:integer; var n:integer);
begin
  writeln('proc2 entered');                   { wolf fence }
  n := l * m;
  global := 42;

  if n > global then
  begin
    writeln('point 1 in proc2');              { wolf fence }
    n := n - 1
  end

  else
  begin
    writeln('point 2 in proc2');              { wolf fence }
    writeln(n)
  end;

  writeln('proc2 done')                       { wolf fence }
end;

procedure proc1;
var i, j, k: integer;
begin
  writeln('proc1 entered');                   { wolf fence }

  for i := 1 to 10 do

    for j := 21 to 25 do
    begin
      writeln('point 1 in proc1');            { wolf fence }
      writeln(i,j);
      global := 23;
      proc2(i,j,k)
    end;

    writeln('proc1 done')                     { wolf fence }
 end;

 begin                       { main program }
   writeln('main program entered');           { wolf fence }
   proc1;
   writeln('main program done')               { wolf fence }
 end.
```

FIGURE 5.5-11a and b: An example of using "wolf fences" in a UCSD Pascal program (a), above, and a portion of the resulting output (b), opposite.

```
main program entered
proc1 entered
point 1 in proc1
121
proc2 entered
point 2 in proc2
21
proc2 done
point 1 in proc1
122
proc2 entered
point 2 in proc2
22
proc2 done
point 1 in proc1
123
proc2 entered
point 2 in proc2
23
proc2 done
point 1 in proc1
124
proc2 entered
point 2 in proc2
24
proc2 done
point 1 in proc1
125
proc2 entered
point 2 in proc2
25
proc2 done
point 1 in proc1
221
proc2 entered
point 2 in proc2
42
proc2 done
point 1 in proc1
222
proc2 entered
point 1 in proc2
proc2 done
point 1 in proc1
223
proc2 entered
point 1 in proc2
proc2 done
```

FIGURE 5.5-11b

BEGIN/END construct or procedure. In the case of procedures, outputting the name of the procedure in the message makes the fence's location easier to find in the source code listing.

Setting Breakpoints

Recall from our discussion of debuggers that a breakpoint is a planned halt in the execution of the program that allows us to examine the contents of variables. More advanced techniques might allow us also to change the values of certain variables, and then resume execution of the program from the point of interruption. In compiled languages, this is usually impossible without the aid of a debugger. Breakpoints can be easily set to interrupt the program, but once this

FIGURE 5.5-12a: Examples of placing explicit "breakpoints" in a Pascal program. In (a), below, a GOTO statement is used to branch to the end of the program. This is how such a mechanism would be handled in standard Pascal. How this is done in UCSD Pascal is shown in (b), opposite, using a special instruction called *exit(program)*.

```
program nonsense(input,output);
label 99;
var global: integer;

procedure proc2(l,m:integer; var n:integer);
begin
  n := l * m;
  global := 42;
  writeln(n)
end;

procedure proc1;
var i, j, k: integer;
begin
  for i := 1 to 10 do
    for j := 21 to 25 do
    begin
      writeln(i,j);
      global := 23;
      goto 99;
      proc2(i,j,k)
    end
 end;

 begin          { main program }
   proc1;
 99:   end.
```

is done, values are no longer available to us. In addition, once interrupted, the program execution cannot be resumed. As a result, the only benefit of the interruption is to output intermediate results using a snapshot device (discussed below), and to halt the program so that it doesn't waste time in executing additional code. In both the BASIC compiler and interpreter, the STOP instruction will cause such an interrupt. In Pascal, a GOTO must be included to branch to a line number just prior to the END statement. Figure 5.5-12 gives an example in Pascal.

In the BASIC interpreter, a stopped program can be restarted from the point of interruption by using the CONT command. This is particularly helpful in breakpoint debugging. In addition, the values of any variables can be printed in immediate or command mode while the program is halted. This technique can produce quick results during preliminary explorations of the code, although the same task is usually accomplished more easily with a snapshot device.

FIGURE 5.5-12b

```
program nonsense(input,output);
var global: integer;

procedure proc2(l,m:integer; var n:integer);
begin
  n := l * m;
  global := 42;
  writeln(n)
end;

procedure proc1;
var i, j, k: integer;
begin
  for i := 1 to 10 do
    for j := 21 to 25 do
    begin
      writeln(i,j);
      global := 23;
      exit(program);
      proc2(i,j,k)
    end
 end;

 begin          { main program }
   proc1;
 end.
```

Developing a Snapshot

The snapshot is probably one of the most widely used debugging techniques extant. This is because it is generally simple to apply, and can point out the course of errors as they are propagated through the code. Some languages have included special snapshot instructions that output the values of all variables whenever the instruction is encountered. This can be somewhat overwhelming in a large program, but the technique is extremely useful, since it allows accurate tracing of values during the hand execution or backtracking of a program.

A snapshot can be as simple as outputting the value of a single variable. This is usually the first approach taken when a variable's value becomes suspect. Figure 5.5-13 shows a BASIC program with some extra PRINT statements that output values of interest along with statement numbers (our wolf fence). This latter item, as well as the names of the output variables, is necessary in order to make the output more readable.

FIGURE 5.5-13a: Using PRINT statements in a BASIC program to provide snapshots of the action. In (a), below, minimal snapshots are used explicitly at particular locations. A partial listing of the output generated by the snapshots is shown in (b), opposite. Note the use of wolf fences.

```
100 GLOBAL = 0
110 GOSUB 310
120 GOTO 999                 ' end of program
310 FOR I = 1 TO 10
311   PRINT "line 311:    GLOBAL = ";GLOBAL
320   FOR J = 21 TO 25
321     PRINT "line 321:    I = ";I;"     J = ";J
330     GLOBAL = 23
340     GOSUB 410
360   NEXT J
370 NEXT I
399 RETURN
410 N = I * J
411 PRINT "line 411:    N = ";N;"      GLOBAL = ";GLOBAL
420 GLOBAL = 42
430 IF N <= GLOBAL GOTO 460
431 PRINT "line 431"
440   N = N - 1
450 GOTO 499
460 PRINT N
499 RETURN
999 END
```

```
line 311:   GLOBAL =   0
line 321:   I =   1    J =  21
line 411:   N =  21    GLOBAL =   23
 21
line 321:   I =   1    J =  22
line 411:   N =  22    GLOBAL =   23
 22
line 321:   I =   1    J =  23
line 411:   N =  23    GLOBAL =   23
 23
line 321:   I =   1    J =  24
line 411:   N =  24    GLOBAL =   23
 24
line 321:   I =   1    J =  25
line 411:   N =  25    GLOBAL =   23
 25
line 311:   GLOBAL =  42
line 321:   I =   2    J =  21
line 411:   N =  42    GLOBAL =   23
 42
line 321:   I =   2    J =  22
line 411:   N =  44    GLOBAL =   23
line 431
line 321:   I =   2    J =  23
line 411:   N =  46    GLOBAL =   23
line 431
line 321:   I =   2    J =  24
line 411:   N =  48    GLOBAL =   23
line 431
line 321:   I =   2    J =  25
line 411:   N =  50    GLOBAL =   23
line 431
line 311:   GLOBAL =  42
line 321:   I =   3    J =  21
line 411:   N =  63    GLOBAL =   23
line 431
line 321:   I =   3    J =  22
line 411:   N =  66    GLOBAL =   23
line 431
line 321:   I =   3    J =  23
line 411:   N =  69    GLOBAL =   23
line 431
line 321:   I =   3    J =  24
line 411:   N =  72    GLOBAL =   23
line 431
line 321:   I =   3    J =  25
line 411:   N =  75    GLOBAL =   23
line 431
line 311:   GLOBAL =  42
line 321:   I =   4    J =  21
line 411:   N =  84    GLOBAL =   23
```

FIGURE 5.5-13b

Finding the right place to put the snapshot statements can be troublesome at times. Locations are usually decided upon by trial and error. However, as with breakpoints, the most common locations are the beginning and end of each block and module. In BASIC, snapshots can be combined with breakpoints to make a very effective debugging tool.

More complex snapshots can be created using subroutines whose sole purpose is to output variable values. Figure 5.5-14 shows an example in BASIC that can be called from any point in the program

FIGURE 5.5-14a: A more complete example of using a snapshot. In this case, a special subroutine (a), below, is constructed that outputs the values of all variables for the entire program. This may be very difficult to create for a large program because of the large number of variables used, but the effort is only required once. Note that the output has been directed to the printer here. This makes the results easier to use to debug the program. Finally, note how wolf fences were created in this case. A partial listing of the execution results is shown in (b), opposite.

```
100 GLOBAL = 0
110 GOSUB 310
120 GOTO 999                   ' end of program
310 FOR I = 1 TO 10
311 SNAPNUMBER5% = 1
312 GOSUB 510                         'Take a snapshot
320   FOR J = 21 TO 25
321 SNAPNUMBER5% = 2
322 GOSUB 510
330      GLOBAL = 23
340      GOSUB 410
360   NEXT J
370 NEXT I
399 RETURN
410 N = I * J
411 SNAPNUMBER5% = 3
412 GOSUB 510                         'Take a snapshot
420 GLOBAL = 42
430 IF N <= GLOBAL GOTO 460
440   N = N - 1
450 GOTO 499
460 LPRINT N
499 RETURN
500 '
501 ' SNAPSHOT ROUTINE
502 '
510 LPRINT "Snapshot #";SNAPNUMBER5%
520 LPRINT
530 LPRINT "GLOBAL = ";GLOBAL,"I = ";I
540 LPRINT "J = ";J,"N = ";N
599 RETURN
999 END
```

```
Snapshot # 1

GLOBAL =   0    I =   1
J =  0          N =   0

Snapshot # 2

GLOBAL =   0    I =   1
J =  21         N =   0

Snapshot # 3

GLOBAL =   23    I =   1
J =  21          N =   21
 21

Snapshot # 2

GLOBAL =   42    I =   1
J =  22          N =   21

Snapshot # 3

GLOBAL =   23    I =   1
J =  22          N =   22
 22

Snapshot # 2

GLOBAL =   42    I =   1
J =  23          N =   22

Snapshot # 3

GLOBAL =   23    I =   1
J =  23          N =   23
 23

Snapshot # 2

GLOBAL =   42    I =   1
J =  24          N =   23

Snapshot # 3

GLOBAL =   23    I =   1
J =  24          N =   24
 24
```

FIGURE 5.5-14b

to produce a snapshot. Including a statement prior to the call to the snapshot that provides a fence address is usually a good idea.

A snapshot routine can get to be quite large if the program uses a large number of variables. It is not always necessary to output the values of all variables, but it is often helpful.

Another factor that makes using snapshot routines difficult is the use of subroutines and local variables. In this case, any snapshot called from within a subroutine would need to know all of the local variables as well as any global ones. This is extremely difficult in a language such as Pascal, where the scope of a subroutine makes generating a generic snapshot routine next to impossible. The only

FIGURE 5.5-15: An example of snapshot routines in a Pascal program. It is much more difficult to produce snapshots in Pascal because of the hard boundary between procedures that creates local variables and parameters. This requires creating a customized snapshot procedure for every procedure we want a picture of.

```
program nonsense(input,output);
var global: integer;

procedure proc2(l,m:integer; var n:integer);

procedure snapshot2;
begin
  writeln('snapshot2 entered');
  writeln('global = ',global,'      l = ',l);
  writeln('m = ',m,'         n = ',n)
end;             { snapshot2 }

begin            { proc2 }
  writeln('proc2 entered');                 { wolf fence }
  snapshot2;
  n := l * m;
  global := 42;

  if n > global then
  begin
    writeln('point 1 in proc2');            { wolf fence }
    snapshot2;
    n := n - 1
  end

  else
  begin
    writeln('point 2 in proc2');            { wolf fence }
    snapshot2;
    writeln(n)
  end;
```

alternative is to create a separate snapshot routine for each procedure within the program, a decidedly inefficient solution. This problem might make any inconveniences of a debugger more palatable.

Figure 5.5-15 shows an example of a Pascal program that has snapshot routines for debugging.

FIGURE 5.15 ***continued***

```
    writeln('proc2 done');                   { wolf fence }
    snapshot2
  end;              { proc2 }

  procedure proc1;
  var i, j, k: integer;

  procedure snapshot1;

  begin             { snapshot1 }
    writeln('snapshot1 entered');
    writeln('global = ',global,'      i = ',i);
    writeln('j = ',j,'         k = ',k)
  end;              { snapshot1 }

  begin             { proc1 }
    writeln('proc1 entered');                { wolf fence }
    snapshot1;

    for i := 1 to 10 do

      for j := 21 to 25 do
      begin
        writeln('point 1 in proc1');         { wolf fence }
        snapshot1;
        writeln(i,j);
        global := 23;
        proc2(i,j,k)
      end;

      writeln('proc1 done');                 { wolf fence }
      snapshot1
   end;

   begin                   { main program }
     writeln('main program entered');        { wolf fence }
     proc1;
     writeln('main program done')            { wolf fence }
   end.
```

6 Documentation

6.1 INTRODUCTION

As stated previously, the chore of documenting even the smallest program is one that most programmers would rather ignore. There seems to be an enormous phobia when it comes to sitting down and writing about a project that may have taken several months to complete. This appears to be the case even among programmers who recognize the need for detailed documentation. Even more alarming, however, are the countless numbers of programmers (and, sadly, managers) who *don't* see the need for spending extra time in documenting a system. It apparently never occurs to them that their creation might need to be analyzed by someone other than themselves, or that even they might forget some of the details of a project after spending several months on it.

Part of the blame for the lack of adequate documentation rests with a company's management. Obviously, if a project manager places high emphasis on documenting a system that is under development, programmers will also emphasize it. Management typically resents the need for documentation, however. This resentment appears to be generated by the feeling that a programmer who is documenting a system is wasting time because he or she is not coding. After all, a programmer is hired to program, not to write. In addition, managers often have a preconceived notion that programmers *can't* write because they are technical types, and everyone knows that technical skills are in direct opposition to communication skills. In other words, if you can program, you probably can't write, and if you can write, you probably can't program.

This all perpetuates the myth that programmers shouldn't write documentation. The end result is that documentation is often given the absolute lowest priority of all aspects of a development project. Also, because of its lack of support, documentation is usually the last part of a project to be completed. This forces documentation work to be rushed, so that whoever is doing it can be reassigned to another programming project. Finally, documentation duties are often assigned to a "technical writer" who probably knows little about computing. While the documents produced in this manner are typically more readable, they are also much more likely to be wrong in some technical detail. The cliché image of a father putting a bicycle together for his kids on Christmas Eve comes to mind. The instructions always seem to have been written by someone who knows nothing about bicycles or, worse, about how tools are used. To make the situation even more horrendous, the instructions were probably translated from Japanese by someone who doesn't know English!

There are a number of other reasons given for not emphasizing documentation. Ostensibly, the most reasonable excuse is that the original programmer(s) will always be around if there is ever any problem with the system. According to this argument, documentation isn't needed because the original programmer will (naturally) be the maintenance programmer (i.e., the one who makes fixes or adds new features to the system). While this may seem to be the most logical approach to maintaining systems, the appeal of this approach quickly deteriorates in the face of certain truths. First, good programmers are hard to keep happy, and are notorious for changing jobs frequently. Second, it should be obvious that a programmer who is maintaining a number of systems he or she previously developed will not have much time left for developing new programs. This almost always results in an unhappy programmer. Finally, even if the original programmer is still around, chances are he or she is so engrossed in a current development project that details about the implementation of previous systems are forgotten. This means that he or she will have to spend as much time (re)learning how the earlier program works as anyone else. So why not give the system to someone else to maintain in the first place? Since this "someone else" is often a junior programmer with little practical experience, the need for comprehensive documentation becomes immediately obvious.

But what about the amateur programmer, developing software for his own needs? The above is an especially seductive argument for the nonprofessional programmer who knows that the system he is developing is going to be used only by himself. After all, he isn't ever going to go somewhere else and leave his systems behind.

There are two very good reasons that the investment in solid documentation pays off even for the amateur. First, an amateur would have the same trouble as a professional keeping track of all the details of a project entirely in his or her head. In fact, you can imagine quite easily that the problem would be many times magnified simply because the programmer is *not* a professional. Anything that can make the maintenance task easier is bound to be a plus in the long run, especially for the amateur. After all, the amateur's real job is something other than programming!

Second, as farfetched as it may seem at the time the system is developed, eventually someone will ask to borrow one of your programs. Without documentation, this would be impossible for a variety of reasons. To begin with, how will the borrower know how to run the system? Will you want to sit and teach the user everything about the system? Will you have the time? For every user who borrows the system? Even more important these days is the prospect that a unique application is actually marketable. Everyone who owns a home computer dreams about striking it rich witb a really super program. The prospect isn't as unlikely as betting on the lottery, so there is plenty of reason for preparing for the possibility. Even game programs need to be well documented for publication. It will become evident that you might have a commercial product if those who borrow your system actually use it routinely. However, no software publisher will even look at your system unless it is cleanly and clearly documented down to the finest detail.

As you can see, there are a number of reasons for spending time and effort documenting your systems. The final detail to be resolved, then, is describing exactly what good documentation is. Unknown to even many professional programmers is that there are at least three distinctly different types of documentation, each with its own special purpose. The most obvious type is **internal documentation.** Most programmers are familiar with this method of documenting the actual code of a program. Its purpose is to provide English translations for sections of code that might be obscure in their function. Unfortunately, not all programmers are taught a uniform method of documenting programs, and there are no standards except those used internally by some companies. Even more unfortunate is that many companies allow their programmers to stop after this first level of documentation. Many professionals are convinced that this type of documentation is all that is really necessary to maintain a program.

Next is detailed **system documentation.** One of the big problems in relying solely upon internal documentation is that it documents only the specifics of a single module. System documentation must look at the big picture of all interacting modules that make up the system. In addition, it is the system documentation that the maintenance programmer must rely upon when an update to the system is needed. This documentation must be as faultless as a map, or the programmer will become quickly and hopelessly lost. Time spent looking for the right place to make a change is essentially wasted. System documentation should make it a simple matter for a good programmer to quickly get his or her bearing and proceed with making changes.

Finally, there is **user documentation.** Even if programmers are cajoled into creating the system documentation, they rarely, if ever, create user documentation. At least, this has been the tradition. Since the advent of microcomputers and the avalanche of interactive applications, however, user guides have become indispensable. In fact, an entire segment of the publishing industry is now given over to producing "How To" guides for every conceivable software and hardware product. There is good reason for this. As noted above in describing the bicycle scene, a product is generally only as good as its user documentation. Thought of another way, a user is not going to use a product if he or she can't figure out how. Good user documentation can make even the most complex software system easier to use. While this level of documentation can often be skipped when the programmer is also the user, even the simplest guide for getting started with the system can prove an invaluable time saver for the person borrowing the program.

6·2 INTERNAL DOCUMENTATION

Having spent a number of years teaching programming on a variety of levels, I can attest to the fact that the single most hated chore of computer science students is documenting what they have done.

Oddly enough, faculty themselves cannot agree on the amount of documentation a program should have. Yet, even when enormous class time is spent explaining the rationale behind neat, clear, concise internal documentation, there are still unbelievers among students. Even more amazing is that when precise instructions for how a program should be documented are given, a number of students still don't do it right! Some just never seem to get the point.

It appears to be a universal trait that even the best students leave documenting their code to the last minute. This is, perhaps, natural, in that there isn't much point spending time documenting something that doesn't work correctly. A beautifully documented program that doesn't work is still worth less than one that works but isn't well documented. However, the objective here is to develop a total package, including all the necessary documentation to maintain the system. Therefore, a useful habit is to get used to documenting code as you go.

As we shall see, this isn't as time-consuming as it first appears. As strange as it may seem, many programs are actually over-documented when it comes to internal documentation. This is because of a misconception concerning internal documentation. Many programmers seem to want to state the obvious and place comments on every line of code. But internal documentation should actually follow the same kind of general development that has been followed all along in the system, i.e., modular.

The main function of internal documentation is to describe the function of blocks of code. This is a natural extension of the structured programming techniques, and is therefore rather easy to accomplish. Documenting the program in this manner can be done piecemeal, in exactly the same way that the modules themselves are developed. When a module has been implemented, take the time to document the code completely before moving on to the next module. This accomplishes two things. First, it makes the entire job of internal documentation less tedious by breaking it up. Second, the documented module is entirely ready to use within the system. The documentation serves its function while you are implementing other modules by making it easier to remember the details of modules already implemented.

Code documentation should make the program easier to read. Therefore, first and foremost, it should be readable. This may seem like stating the obvious, but it is amazing how often this advice is ignored. Abbreviations should be avoided except where absolutely necessary to save space. Jargon should likewise be avoided at all cost.

Obviously, this is not always possible, since some terms uniquely describe an activity or item.

Internal documentation will take on an outline form, since it will follow the natural lines of modules and blocks of code. This will serve several functions. First, it means that the documentation will be more or less standard in form, making it easier to write. Second, the documentation will be easier to read because of the standard form. Finally, following the outline form will make blocks of code easier to identify, making maintenance simpler.

Disadvantages

As noted earlier, there is a tendency to use too many of the wrong kind of comments in a program. In this way, comments get out of hand and do more to obscure the function of the code than to illuminate it. Comments should not simply translate an individual statement into English, but should be used to identify whole functions being performed in the code. Only occasionally should a single line be given its own comment. This is discussed later in this chapter.

In addition, the format of the comments themselves can make the program difficult to read and, therefore, understand. Publishers have known since the beginning that the style of presentation can make all the difference in the world between a book being a useable tool and entirely unreadable. The same rule applies to comments in a program's code.

While usually not a problem, comments can occasionally cause a program to be too big. This is particularly troublesome in microcomputers with small memory sizes or small mass storage capacities, especially when programming in BASIC. Because BASIC is usually an interpreted, not a compiled, language, the entire source code of the program (the BASIC statements themselves) must be loaded into memory at one time. The BASIC interpreter must then look at every line that is numbered, including REM (remark or comment) statements, during the execution of the program.

Note that every character in a comment would take up exactly one byte either in memory or on a floppy disk. When you consider that comments can more than double the size of a program's source code, and that even simple programs can quickly become large, the problem becomes obvious.

Even more surprising, perhaps, is that comments can make a program written in an interpreted language such as BASIC run quite a bit slower. This is because, as noted above, every line in the source code must be looked at by the interpreter for program statements. This is true even if the line only contains a comment. And since the interpreter has no way of remembering which lines contain comments, it would have to look at, and then ignore, lines of comments every time it came to them! This ultimately causes the execution speed to decrease.

The only resolution to the above two problems is to remove the comments from a program before you run it. "Why bother to put the comments into the code if I'm just going to take them out again?" you might ask. Well, certainly this would be a futile gesture if the comments were lost forever. However, there is an automatic technique for removing comments from most BASIC programs. It involves creating what is sometimes called a **preprocessor** program that creates a second program file with the REM statements removed. This leaves the original, commented file intact. Any changes made to the program are made to the original, commented program. This source file is then run through the "compression" preprocessor to create a new uncommented file that is then executed.

The appendix gives such a preprocessor program. It is included as a complete example of a fully documented, ready-to-use piece of software.

The final disadvantage of internal documentation is that keeping all of the internal documentation up to date can be tedious at best. It requires diligence to make absolutely certain that everything is accurate. Any inaccuracy or ambiguity in the comments can cause countless wasted hours for a programmer who reads the comments, assumes them to tell the truth, and only finds out much later that they were wrong. While certainly it is not simple to keep the documentation up to date with the code, it is absolutely necessary. What good are comments if you can't trust them?

"Self-documenting" Languages

Pascal is often cited as one of a class of programming languages called "self-documenting." What other languages are in this class is generally not stated, but I suspect that any structured language that allows long enough variable names would fit. Some versions of BASIC might, therefore, be included.

The idea is that a well-written program implemented in a modern, structured language using accepted programming techniques does not need additional comments to clarify the code; the meaning of the code becomes self-evident. While this idea is good, I have yet to see it flawlessly implemented. For example, one of the programming techniques helpful in achieving self-documenting programs is to use meaningful variable names, so that the usage of any variable is instantly obvious. I have previously (see section 3.2, Implementation Guidelines) discussed the desirability of this technique, as well as its pitfalls. The length of variable names needed to impart the total meaning of a variable becomes impractical. Imagine needing variables called *OLD-ACCOUNTS-RECEIVABLE-TOTAL* and *NEW-ACCOUNTS-RECEIVABLE-TOTAL*. The natural impulse is to abbreviate to something like *NEW-AR-TOTAL*, but now the variable has lost some of its instant recognizability.

In addition to this problem is the fact that what is obvious to one programmer may be totally obscure to another. We all bring our own set of assumptions to any task (as was discussed in Chapter 4, Human-Engineered Programming). Therefore, no code, even in the simplest programs, can ever be truly "self-documenting." While any technique that makes a program's code more readable is certainly worthwhile, no one technique can be relied upon to provide everything that is needed in the way of documentation.

Program Header

A standard format of comments at the beginning of every program provides you with a ready, quick reference to what the program is about. It also contains some information that you will come to rely upon when performing any maintenance. For BASIC programs, I always reserve line numbers up to 99 for the program header comments. While I seldom need all of these lines, it is nice to know they are available for expansion. Obviously, for languages such as Pascal that do not use line numbers the way BASIC does, this is not necessary.

Refer to the program listing in the appendix during the following discussion.

The **program name** should be clearly stated at the top of the listing. This name will also usually be used as the file name of the source code in mass storage. However, since some file systems only allow a few characters for a name, abbreviations must sometimes be used. In these cases, be certain to use both the abbreviated and the spelled out forms of the name at the top of the listing. It also doesn't hurt to be extra cautious and specifically identify which is the file name. Finally, where more than one file name exists relating to a program, as in the case of a compiled program having one file for the source code and another for the object code, he certain to include specific references to these.

It is often customary to number slightly different versions of the same program. This is done as insurance during the implementation or maintenance phases. When changes are made in a program, a copy of the current version of the program is made first. Then changes are made, and this new version is given a new version number instead of an entirely new name. The old version is thus preserved as a backup copy in case anything goes wrong during modification, such as the file getting accidentally deleted. Different versions are also sometimes customized for particular purposes.

Usually only one or two old versions of the program are kept, the more outdated ones being deleted. The version number typically makes up the last one or two characters of the program name. However, since this shortens the possible file name of the program even more, the version number is sometimes simply listed in a comment at the beginning of the source code, even though this makes differentiating various versions of a program more difficult.

Next, the name of the author(s) should be given, with a copyright notice. While the copyrighting of software is still not universally recognized as legal, the point is generally made that this is indeed your program and that only you can authorize its use. Most people will give it appropriate respect.

A synopsis of the purpose of the program is generally given next. While the name of the program should give some clue as to its function, this paragraph should make that function quite clear. Naturally, it is sometimes difficult to summarize a large or complex program in just a few sentences, but details can he saved for the rest of the program comments.

A list of any input or output files used by the program makes it easy to quickly locate the files without having to look through the entire code.

An important element of the program header that is often overlooked is what is usually called a variahle dictionary. This dictionary is based on the one developed for the algorithms (see section 2.3, Algorithm Design), although there may he more variables introduced during the implementation coding. If the dictionary is too large to fit in the program header comfortably, include it in the system documentation instead and put a reference to it in the header.

This might seem a rather large joh, since even a simple program can use many variables. However, remember that your code is modularized. This breaks up the use of variables, so that each module really only references a few. This makes the variable dictionary much more manageable.

Next, I've always found it helpful to keep a log of any changes I've made to a program. This makes it easier to pick up where I've left off when I stop work on a program for a while. Also, since not all the changes I make work out the way I had in mind, this log lets me backtrack to find errant changes. One of the many corollaries to Murphy's Law is that the worst bug in a program never shows up until the program has been in use at least six months. With this in mind, it is easy to see why one would want to know what the last change made to the program was. Maybe it was that change that introduced the bug!

To provide this log, I usually include a list of all the major changes I've made to the program, and the date each change was made. Many professional programming systems on large-scale and minicomputers provide a software package to automatically keep track of changes to the source code. Since this is a rather expensive trick, this type of product is not (yet) available for microcomputers.

But what changes should you include in this log? Well, it really isn't worth the trouble to list every time you fixed a typo. However, any time new features are added to the code, I include them in the list. Also, any patches made to the program to fix bugs discovered after you initially "finished" the program should be included. The log section of the header is really only needed after you start using the first version of the program and discover that either something doesn't work right or you wish it worked differently. The log comes at the end of the program header because it will naturally grow during the life of the program.

Subroutine Header

Subroutine headers can be treated like miniature program headers. Since they are often modules themselves, subroutines need descriptions very similar to those for the program as a whole.

First, place the subroutine name in a very prominent comment. For a language such as BASIC that does not have true subroutine names, giving subroutines names makes them much easier to remember when reading the code. For languages such as Pascal that do allow names for subroutines, repeating the name in a comment might appear redundant, but the name comment can also create a visual separation of the subroutine code, making the code much easier to locate. Even more preferable, for a hard copy listing of a program, is for each subroutine to begin at the top of a new page, thus further separating the code visually. Obviously, there is no way to cause this to happen when listing the code on a monitor.

Next, list the input and output parameter names for the subroutine and give a short description of each. This makes the parameters more visible, which is especially necessary for BASIC, which has no formal parameter mechanism. This procedure can generally be skipped for any language that does allow parameters.

A short description of the function of the subroutine should be next. Since a subroutine should have a very specific function, and usually only one function, this description should not need to be long.

Finally, a variable dictionary for any local variables should be given. There is no point in describing these local variables in the main program variable dictionary, since they either are not used elsewhere in the program or they have an entirely different meaning from their use in the subroutine.

Don't be afraid to use blank spacing to help make the comments more readable. Other formatting techniques such as dashed lines or asterisks to highlight certain comments are also very useful.

As noted in section 3.4, Effective Use of Subroutines, one suggestion for BASIC subroutine headers uses line numbers ending in 00 to 09, with the subroutine line number ending in 10. While this might not seem like enough room for all the above comments, it usually is. Barely.

Block Header

Code was implemented in blocks according to the level of the function. Each block had certain characteristics, such as one way in, one way out, and one function. This being the case, it should be obvious that we would want to document each of these blocks in the code. This usually only requires some type of header paragraph to give a general description of the function of the block. In addition, the block header can be used to discuss any special programming technique that might have been used in this section of code.

Since blocks of code don't usually have names or local variables associated with them, these kinds of details can generally be ignored in the comments for blocks. However, it is again a good idea to use comments to separate the code and make it more readable. Use white space or fancy lines to make the blocks easy to find when skimming the code.

Line Comments

For the most part, the previous methods of internal documentation eliminate the need for detailed comments on nearly every line of code. In fact, if the "self-documenting" nature of a modern language is taken advantage of, no additional line comments should be needed. However, there is almost always some little obscure piece of code that requires just a little more documentation to make everything abundantly clear.

Line comments should be more than the English equivalent of a code statement, however. For instance, if we had the following line in a BASIC program,

```
620 I = I + 1          'increment I
```

it should be obvious that the comment lends no additional meaning to the code. Chances are that the meaning of any line as simple as the one above, considered individually, is so obvious that it is pointless to comment on it further. It is usually only when statements are taken collectively that their meaning is obscured.

However, it is occasionally necessary to employ devious tricks when programming, necessitating a special line comment. Consider the following line of code:

```
460 IF A >= 65 OR A <= 90 THEN A = A + 1
```

An appropriate comment should be added to make the function clear. For example:

```
458 REM if A's ASCII code shows it to be a character,
459 REM change A to be the next character in the alphabet
```

This would make understanding line 460 much easier, especially for someone not entirely familiar with the ASCII code. Naturally, line comments should be kept to a minimum, as they have a tendency to get in the way. However, when in doubt, go for clarity and add the comment.

6·3 SYSTEM DOCUMENTATION

For some reason, many programmers figure they are finished with the program once they complete the internal documentation. What they fail to realize is that whoever is assigned as maintenance programmer for the new piece of software will need a great deal of help in understanding how the program was developed. The biggest aid to the maintenance programmer is a comprehensive set of documentation about the design and implementation of the system.

As you will soon see, the task of preparing this documentation is not as difficult as it might first appear. The majority of the detailed documentation has already been created if you followed the other techniques outlined in this book. It is mostly a matter of collecting final forms of each of the techniques used. There will undoubtedly be a few new things, but they will be fairly painless to create.

Since the main purpose of system documentation is to help the maintenance programmer in his or her job, this documentation will be very technical in nature. It assumes that the reader has the proper technical background to make sense of it. This does not mean that you shouldn't be careful in how the details are collected or presented, however.

It does mean that you can assume that the person reading the material is probably not the end user. In today's world that may be a somewhat hasty assumption, but it is close enough to the truth to be useful. If the user is really the one reading this level of the documentation, then the user is probably sophisticated enough to handle the technical detail. If not, the user shouldn't be reading this level of the documentation.

The next section discusses what should be included in a comprehensive user's guide. It is important to remember that there may be a distinct difference between the maintainer of a program and its user. Therefore, different documentation guides should be developed based on that difference. Attempting to provide both levels of documentation in a single guide will usually only result in neither the maintainer nor the user getting what he or she wants or needs.

Specifications

The **specifications,** or **specs,** of a program are the description of what the program is supposed to be. These can be extremely formal, developed in their own special-purpose language by a systems analyst. In professional environments, the analyst's job is to prepare these specs in great detail, to the point of doing much of the initial design work. The specs are then turned over to one or more programmers who implement the design into code. This whole area of specifications is getting so formal these days that much research is devoted to getting the computer itself to write a program from the formal specifications. Actually, what this means is that a new metalanguage is being developed that will make it "easier" to write programs. There doesn't seem to be any imminent danger of programmers losing their jobs, however.

Specifications can also be much more simply thought of as an informal description of what the program ought to do. Even when working with this informal description, however, great care should still be taken that the specifications are clear, accurate, and unambiguous. This was all discussed in some detail in section 1.2, Defining the Application. For the purposes of documentation, specifications can be broken into three parts. Refer to the System Documentation section of the appendix throughout this discussion.

General Description

A simple verbal description of what you want the program to do for you should be included in the system documentation. This was the very first thing you developed when you decided that there was some job you would like the computer to do for you. The description should include an explanation of the various functions you want performed. In addition, it should include a very general discussion of what data the program is going to operate upon. This doesn't have to be very detailed. Design and implementation details will be presented later. This description is only a quick summary of the purpose of this piece of software.

The description could also include an example of your original thoughts about the expected output. If the output was to be some type of reports, a rough sketch of what those reports might look like should be added. Naturally, what they really look like might differ, even significantly, from your original idea. However, there ought to be some record of the evolution of the program so that it can always be verified that you ended up with what you wanted. This is especially true if you are developing a program for someone else, and particularly if you changed something from the original program definition.

Hardware Requirements

Many programs have special hardware requirements. The following items should be clearly stated in the documentation:

1. Type of computer
2. Amount of memory required
3. Type and amount of mass storage required (e.g., tape or disk)
4. Special input or output devices
    a. color graphics
    b. joysticks
    c. light pen
    d. printer
    e. electronic test equipment (e.g., voltmeter)
    f. modem
5. Special settings or characteristics of the I/O devices
    a. dot matrix printer with graphics or letter quality
    b. switch settings (e.g., baud, half- or full-duplex) for modem

Most of the information in this list is self-explanatory. The list should include any hardware characteristic that the program depends upon. This information will also be included in the user's guide, probably in more detail. Including it here ensures that the maintenance programmer is aware of all the hardware pieces that the program needs. Never assume anything is too obvious for the list. For instance, it may seem entirely unnecessary to include the fact that your program requires color graphics, because (obviously) every microcomputer system comes with this as standard, right? Even the cheapest microcomputer has color graphics. Well, it may be surprising to you that not only graphics are optional on such systems as the IBM PC and the DEC Rainbow 100, but color is also. If your program depends on colors in order for the output to make sense, then this obviously has to be specified as part of the hardware requirements.

Software Requirements

As with hardware requirements, you should provide a list of all special software considerations that are necessary in order for the program to execute properly. This lot can be broken into five categories: operating system, languages, utilities, commands, and files.

OPERATING SYSTEM First, under what operating system was this program developed? This is easily seen in software that specifies CP/M or MS-DOS for the operating system. In these cases, the first consideration is that the operating system in use is compatible with the format of the floppy disk used to distribute the product. For instance, a floppy disk formatted under CP/M cannot be read using any other operating system without a special utility program.

The second reason for specifying the operating system is that the programmer may have, in fact, used special features that are available only in that particular operating system. CP/M, for instance, has built its reputation, in large part, upon the ability of the application developer to enter the operating system in very specific ways. This provides the developer with some ready-made routines that perform a number of functions, making the developer's job simpler. Since an application developed using these special operating system routines depends on the availability of the routines in order to make the application executable, specifying the operating system used—and the version of that operating system—is necessary in the documentation. Be sure to include references to the special features of the operating system the program uses, and to operating system manuals that describe these facilities in detail.

LANGUAGE PROCESSORS The next question to be answered is, What language was the program developed in? Was it BASIC, Pascal, assembly, or perhaps even several different languages? Next, what version of the language(s) was used? For instance, on the IBM PC you have a choice of three versions of BASIC, and they all come standard with the system. In other cases, a third-party compiler or interpreter might have been used. An example is the BASIC interpreter available from Microsoft, Inc., for most major microcomputers.

It is extremely important to know not only the vendor of the language processor, but also its version. For instance, the BASIC in this book was developed using Microsoft's MBASIC-86, a version available for the DEC Rainbow 100. While a program developed using one language processor might actually work using a different processor, usually some amount of conversion is necessary. Giving details about the original processor helps define what characteristics, functions, etc., the processor might expect to find in the user's program. In addition, it is helpful when converting to be able to refer occasionally to the original language manual.

Finally, the documentation should specify the exact options used when the program was compiled or interpreted. Under BASIC, this would include any options, such as the number of files, that should be specified when entering the interpreter. An example of this is found in the way in which the MBASIC-86 interpreter is entered for the DEC Rainbow 100. For a compiler, it is still important to specify the options used to generate the object code, even though these options are not "vital" for the execution of the program as they are for the BASIC example. Nevertheless, the options can influence how well the program ultimately executes. This is because many compilers have more than one option that changes exactly how the code is compiled. Refer to the user's guide of the compiler you are using for more detail on these options.

UTILITIES Next, are there other utility programs or external routines that this program needs to use? A utility program is one that was developed to be executed by an application program which depends on the function(s) supplied by the utility. An example is a sort program. Here, the application program doesn't need to actually perform the sort, but instead calls the utility sort program to do it. In this way, the application programmer is freed from actually having to design and implement a sort routine.

The **external routine** is similar to the utility, but instead of being an entire program, it is usually a subroutine that performs a small task. The UCSD Pascal system supports a number of these routines to perform functions such as clearing the video screen. External routines are also any routines developed by the programmer and added to a **software library** to be used whenever needed. This keeps the programmer from needing to reinvent the routines every time they are needed. These routines have to be fairly general, and only save time and effort if they are actually used somewhat frequently when developing programs.

The entire area of utilities is a natural outgrowth of the modular method of programming. Utility routines form preconstructed foundations on which other programs may be based.

Finally, the utilities documentation should include any parameters that are necessary to be passed to the utility or routine. This may not be known specifically, but even a general description can be helpful. Include specific references to the user manuals for any utility used.

COMMANDS This area of documentation should include any setup commands that the user must issue before the program itself can be executed. While ideally the program should perform this function itself, perhaps with some direction from the user, this may not always occur. For instance, the IBM PC has a MODE command that changes various characteristics about the screen, I/O ports, etc. This command may not be accessible from, say, Pascal, so the user would have to issue it himself if anything needed to be changed before executing a particular program.

Details about the options that should be specified when issuing these commands need to be included. In this way, all the details about how the task must be performed are contained in one, comprehensive, easily referenced document. Cross references to any manuals discussing the particular commands used are also helpful.

FILES Many systems require a standard set of data files in order to execute. For example, a payroll program might need a file of employee records that would include every employee's name, identification number, and working wage. Another file might contain details on the number of hours each employee worked in a given period. A third file might contain details about how taxes and other deductions are to be calculated. In a case such as this, a simple list of the names of all required files and a short description of each file is sufficient. More details can be included in a later section about the format of the file records. Not every program has a fixed set of files that it needs.

In many cases, the program will request a file name from a user. For example, in the compression program given in the appendix, the program asks for the name of the file that contains the program to be compressed. While the exact nature of the file is not known, at least general characteristics can be presented. For instance, for the compression program, the source BASIC program must be stored in ASCII format, not the usual tokenized form. This information is vital, since the program will not work properly without the right type of file as input.

Design Details

This section of the documentation is very straightforward. It is simply a collection of the various pieces developed during the design phase of the project. A few comments might be added here or there for the sake of clarity, but generally the original lists, charts, tables, or diagrams are sufficient. The following items should be included:

1. data definition and flow diagrams
2. HIPO charts
3. the basic algorithms
4. flowcharts
5. general data structures

In the case of data structures, only a general description is necessary. More detail will be added in the next section of the documentation.

Implementation Details

The previous documentation is quite simple to collect, since it has already been produced during the design phase of the project. It represents the minimum amount of system documentation needed in order to maintain a program. Adding certain details about the implementation of the program, however, can make maintenance even easier. Unfortunately, implementation documentation is usually overlooked as being unnecessary, even though it can save the maintenance programmer hours of searching through program listings or manuals for some obscure detail.

File Descriptions

In the previous section of the documentation we included a general description of the various data structures used by the program. This would include internal structures such as arrays or tables. In this section, a detailed description of every "external data structure," i.e., file, should be given.

To begin with, use the details of the previous documentation sections to create a general description of the file, including information developed during the initial design using the data dictionary. Next, give specific characteristics of the file. For instance, list the following items:

1. the approximate size of the file
2. its frequency of use
3. any special characteristics, such as a write protection mechanism or password

4. the order in which the file is maintained
5. the key(s) of that order

Next, list any other programs that might have access to this file. This provides a handy cross reference for your data and may later help you in tracking down how particular data was placed in the file. Occasionally this exonerates your program from having messed up the contents of the file.

Finally, but most importantly, provide complete and very detailed **record layouts** for every file used. A record layout is a byte-by-byte definition of every possible format a record might have in a particular file. The easiest and most straightforward file to create record layouts for is one that contains a single record format of a fixed size. This kind of layout can be easily derived from the data dictionary, since the details about the type, format, precision, etc., of a data item are fully described here. This is especially true for character type data, since there is a one-to-one correspondence between a character and its internal representation. This is even true for numeric values that are not "true" numbers, i.e., that are never involved in calculations. Examples of such numbers are a telephone number or a student identification number.

But what if true numeric values are to be stored as part of the record format? Certainly we need to be able to store regular numeric values. This indeed poses a problem when describing the record format, unless the programming language you are using allows you to specify an exact size for the numeric value. COBOL and PL/I, for instance, require you to specify the size of every number to be stored. Pascal, on the other hand, doesn't allow you to do that at all. In the case where the size of the number has been specified, naturally include this information as part of the record layout. When this is not possible, simply describe the item in general terms, including its data type (i.e., character, floating point number, integer, etc.).

Some files have more than one type for records that could be a part of the file. This requires that you specify the record format for every type of record the file might hold. It is even more helpful if there is some indication of how one record type is distinguished from another within the program. Perhaps the records come in some sort of order, or have related records associated with them in some way. Another possibility is that special records are located at specific places in the file, such as at the very beginning or end. These records might hold some general information about the file itself, such as the number of records it holds.

Up to now, I've purposely avoided mentioning file formats when using BASIC. That's because BASIC (and some other languages) has a file system that is entirely variable, and does not rely upon the record concept to store information. Instead, every individual item that is to be read from or written to the file must appear on the I/O statement. The INPUT statement must contain a variable name for each item stored within the file as a record. The problem is that BASIC allows us to read or write this same group of data in several different ways, using more than one INPUT statement.

In addition, the size of a "record" in BASIC is entirely variable in the case of sequential files. It depends on the type of data being written, and does not require that each item have a specified length. In addition, numeric values are stored differently depending on whether they are floating point numbers, integers, or something else (some BASICs, for instance, allow a one-byte integer value).

Because the file structure in BASIC is essentially free format, it is rather difficult to describe in terms of records and specific formats. However, it is important that some record be kept on the data and its format within a file. Therefore, I recommend pretending that files are indeed record oriented (see section 3.2, Implementation Guidelines). Even if the exact length of every record format is not known, at least a general description of the various fields within each "record" can be provided. This is especially true in a language such as BASIC that does not require the programmer to specify a format for records within some declaration statements. Otherwise, one might never be able to decipher the actual format of the data in a file from a BASIC listing, since the statements that read or write the data can literally be spread across the entire program.

Special Programming Techniques

I can still recall with great vividness the first big system I was asked to maintain. I was working for a very large corporation with an enormous computing staff, with several huge computers and hundreds of programs. The program I was to maintain took raw data from the computer as it was operating and turned it into statistics about how well the system was performing.

About a week after the system was given to me, a change in the program was requested to allow the program to gather some additional data from the computer. I diligently pored over the documentation and listing for the program, trying to understand the algorithm the original programmer used. After several days of study, I still couldn't make sense of what the program was up to. Luckily, the original programmer was still working for the company in another department, so I went to see him.

After my brief explanation of the nature of my problem, he began poring over the listing of the program himself, trying to refresh his memory about just what he had written. After more than an hour, he finally remembered the trick. "Of course. I used a finite automaton as the basis for the algorithm," he declared, proudly. I vaguely remembered studying such beasts when in school, but had never seen one used for a real application. "Yes," he continued, before I could question him further, "I read about them in an article about the time I got this assignment and wanted to try out the technique. So I wrote the whole program as one giant finite automaton." That's all I ever got out of him about how the program worked. He couldn't remember what the article that had set him off had been, or even what journal it might have been in. In addition, he couldn't even explain to me, at least to my own satisfaction, how it worked. After wasting both his and my afternoon, I finally resorted to digging through the company as well as the local university library looking for references on this technique. It took me several days to acquire enough background to muddle through the modification of the program.

There are a couple of morals to this story. First, never use a technique just for the sake of using it, even though it might indeed work. You should always try to match the technique to the situation. That is, there should be a good reason for using a particular approach to solving the problem. The above-mentioned programmer could give no particular reason for using the finite automaton technique other than that he thought it would be fun.

Perhaps more important, any special techniques that might be used in a program should be heavily documented. This should include your own explanation of the theory of the technique, as well as a detailed description of how the technique was used in the application. List any references that you may have used in learning or developing this technique, and be as specific in these references as possible. This might even include a copy of an article that could later be hard to find in a library.

Other techniques that should be included here are any hardware-dependent programming tricks that another programmer might find somewhat obscure. For instance, in some applications that use a particular video terminal, there might be specific functions or codes that change the characteristics of that terminal. Perhaps there is a special code that clears the screen or puts the terminal in reverse-video mode. While these tricks will obviously be pointed out and explained in the internal documentation, it never hurts to make everything abundantly clear by mentioning them in the system documentation as well. As in previous cases, this extra documentation could make the job of converting the program to some other system much easier.

Error Handling Methodology

In Chapter 4 we discussed the need to provide extensive error handling capabilities in any sophisticated system. The various methods employed for this function should be described in detail on a technical level. The actual error messages themselves and their implications will be a part of the user guide.

In the system documentation, the interest is in providing details on *how* errors are handled. This is needed by the maintenance programmer for reference when making changes to the program. One should never have to search through the actual code to find all the ways something has been done. Any new code should include the same kind of error detection and correction as the original code. This documentation is insurance of that consistency.

Limitations

Nearly every program I have ever seen has a limitation of some kind. This usually results from fixed definitions within the program that imply some kind of size restriction. For instance, the size of an array defines a fixed maximum for the number of items it can hold. Other limitations might be restrictions on various inputs, the sizes of files, or the number of a certain item. An example might be that a compiler or editor will operate only on files up to a certain length. In a checkbook program, there might be a limit on the number of checks or deposits allowed in a month. A comprehensive list of such limitations helps the programmer to determine if one of these limitations is causing a current problem with the program.

Future Extensions

Almost certainly, during the development of the program you will be struck with many wondrous visions of how this program might be improved. However, you should always avoid the temptation of abandoning all your work up to that point in order to include this new wonderful feature. The reason to avoid it is that you will never get *anything* finished if you are constantly going back and revising the specifications of the program. Rather than do this, you should simply keep track of all the features you would like to add to the system (at a later date). It is far more important that you should get *something* operational than that the program be perfect the first time. After all, the whole point of doing all this "extra" work using structured techniques is so the program is easily modified!

Listing

Don't forget to include a listing of the latest version of the program's source code in the documentation package. Take the trouble to get a good, clean, clear listing. Put a new ribbon on the printer if the output is difficult to read because the printing is light.

Also, if your compiler, interpreter, or other utility packages offer you extended listings beyond just the code itself, be sure to include these. An example would be **cross reference tables** of all variables used in each module. These provide a listing of every variable used in the program, and every line in the program where each variable is used. Other useful options are nesting level indicators and labels at the beginning and end of every block of code. Finally, make use of any type of listing formatter, usually called a pretty printer, that might be available. This formatter automatically repositions the code in the listing to show blocks, separate subroutines, etc. If you have taken care when writing the code, this might not be necessary. However, it is still a good idea in order to see if the computer thinks the program contains the same blocks you do.

Test Documentation

Chapter 5 presented the gory details of how to test a program to ensure that it does what it is supposed to do. Test documentation should include details about the various tests that were performed. The purpose of the documentation is twofold. First, if a problem should crop up in the future, this documentation can provide some answers as to why the bug didn't show up during the test phase. The tests can then be modified to check this area of the software.

Second, test documentation provides the means for retesting the software after any other type of modification is made. It helps to duplicate the original test conditions, thus ensuring consistency in the test procedure. Also, why should the programmer have to think up all those complicated test cases all over again?

Test Data

Any data that was used to test the software during its original verification should be included here. This will usually include samples of the various inputs and data files used. The documentation may also refer the programmer to files containing test data. In large computer shops, **test data generators** may be used to help the programmer to produce enough data to adequately test the program. This data is usually then preserved in a test data library for use during the maintenance phase.

Test Runs

If at all possible, the output results of the test runs should be collected and included in the documentation package. This might not be possible when the program is interactive in nature. However, samples of any printed reports should certainly be retained. In addition, listings of files that were generated by the program during the tests should be included. Any listings should be annotated by hand. This should include any details about special conditions that may have been used during the test.

System Relationships

Since this program might be only one of several in a large system, it is important that the relationship of this particular program to any others within the system be fully documented. Full system diagrams should be included in the documentation packages of every program within the system. This ensures that this important information is never far from hand.

In addition to general charts showing the relationship of various components of the system, system-wide data flow diagrams should be included in this package. Using these, one can see, for instance, what effect changing the format of a particular file might have on the rest of the system.

References

A complete list of all other manuals, articles, or books used while designing and implementing this program should be provided. This makes it more likely that someone else will be able to recreate your steps if the need should ever arise. Include the references cited elsewhere in the documentation so that this list provides a comprehensive reference.

Error Reports

The final section of the documentation should be devoted to describing any problems that have been experienced by users of the system. Such problems could arise from two sources, the user and the program itself.

The user might have misunderstood how a particular feature of the system was supposed to work. In this case, it usually indicates that either the user didn't read the documentation carefully, or the documentation was somehow deficient.

If the user was at fault, there is not much that can be done except to refer him or her back to the appropriate section in the documentation. If a number of users have a similar problem, however, it is usually an indication that there is something missing from the documentation. If this appears to be the case, the documentation should be modified to take care of the specific problem. This can be done by circulating addenda for the documentation manual to the users. A better, but more time-consuming, approach is to rewrite the section of the documentation dealing with the feature that caused the problem.

When a real program bug is discovered, however, it requires modification of the program itself. This is obviously a much more serious matter. In order to keep track of all changes to the program, careful notes should be kept by the maintenance programmer. This allows the programmer to reconstruct the history of every change that has ever been made to the system. Such a history is vital in the event that one of these "fixes" is, in fact, the source of a new set of bugs.

This **error report** should include a detailed description of the symptoms that the user noticed. This description is helpful to future programmers in the event similar symptoms occur at a later date.

The report must also include details about the cause of the error.

This is different from the list of the symptoms. The symptoms are from the user's viewpoint, while the cause is given from the programmer's viewpoint. The report should reference the source code and other system documentation as appropriate. Documentation so referenced should be copied and included in the report, in case the referenced documentation sections are later changed.

Next, the changes that the programmer makes to the program and documentation should be listed. This should include the programmer's explanation of why these particular changes were made. Also, the changes to the source code should be given in minute detail, usually by listing the source code that has been affected.

Finally, information about how the program was retested after the changes should be given. This should include the specific tests done to verify the fix, as well as details about any regression testing performed.

6·4 USER DOCUMENTATION

One of the biggest mistakes that programmers make when writing documentation is that they assume users have the same knowledge about programming as themselves. In this fantasy, all users instantly know what to do with the system because they (naturally) have the system documentation. After all, the system documentation gives all the gory details about how the software was designed, implemented, and tested. Indeed, there are many descriptions of what the program does, the hardware and software requirements, and details about the files the program will use. Isn't this enough?

There are two fallacies with this fantasy. The first is the assumption that the user will have access to the system documentation. Recall that the main function of this level of the documentation is to support the maintenance programming task. For probably 99.99 percent of all software, the user is not the same as the maintainer. This is certainly true of commercial software products. In the case of large computer systems, a software vendor charges a fee to provide this maintenance to customers. The customers themselves do not (usually) make changes to the software they purchase.

If you are the one supplying the software package, you don't want customers to monkey around with the program. If they did, and something later went wrong, you couldn't be sure if the cause was a bug in the original program, or one introduced by the customer's change. Indeed, in most cases customers are not even given the source code of the software.

One might be tempted to think that this, seemingly like everything else, works somewhat differently in the microcomputer field. After all, microcomputing is notorious for its informality, its comradery, and its devil-may-care reputation. In addition, the field is still very young. Less than a decade ago you needed to have an advanced degree in electrical engineering to use a microcomputer. Just five years ago, you still needed to be an accomplished programmer.

In the "old" days, the technical types would gather regularly to discuss the latest advances in the technology, discuss new ideas and approaches, and pass around their latest programs. Everyone shared everything. And since everyone was technical, it was natural to share listings. Everyone understood. (Well, almost everyone.) And commercial software closely resembled what was traded at the club meetings, mainly because the first software vendors were garage-shop operations started by some of the same people who attended the meetings.

Before long, however, less technical users began showing up at the user club meetings. These new users demanded more details about how to use a piece of software, and fewer about how it was built. More and more, users were moving away from the programmer end of things. They became consumers.

Today, the computer is just another electronic consumer product, about on par with microwave ovens, digital watches, and calculators. Does any consumer product come with technical details about how the product was designed or implemented? In the glove compartment of any new car is an "owners manual." In it you will learn about the various knobs, switches, and gauges in the car. Typically, however, you will not learn how to do even the simplest maintenance. Perhaps there might be information about proper tire inflation, or where the oil should be added. There is seldom anything about tuning the car's engine, however. And there certainly will not be anything about tuning the carburetor. While thousands of home mechanics are willing to risk life and limb by installing new brakes, the vast majority of car owners leave such technical details to professional mechanics.

And so it is with today's microcomputer market, or for that matter, the computer market as a whole. It has long been known in the professional computer field that users need only know how to *use* the software, not change it.

Thus, the second fallacy is that users *need* to know the technical details about the software implementation. There is a somewhat dangerous tendency to tell users more than they need to know. However, it is sometimes a fine line between telling them too much, and not telling them enough. Doing either is obviously a problem, and even the middle ground can occasionally be shaky.

One of the hidden problems of trying to supply adequate user documentation is that the level of sophistication of various users of the same product might differ significantly. For instance, I as a computer scientist probably don't need to be told some things about a word processing package that a secretary would. For a user like me, a lot of assumptions could be made in the documentation. On the other hand, the secretary would likely need a great deal of detail about how the computer actually works, especially if he or she has never used a computer before. The concept of files and memory might be a little difficult for him or her to get used to, whereas I would have no problem.

Striking a Balance

The problem for the documentation writer, then, is that the documentation must suit all users equally in terms of depth, presentation, and technical content. This is similar to what was discussed in Chapter 4 about the program interface and the user. Addressing the need for an interface that adapts to the user's level of sophistication is one of the foundations of a user friendly system.

Unfortunately, it is not as easy to provide the various levels of information in a printed document as in a program. The medium of print is too intractable. Yet, it must be accomplished in some way. The main reason that the microcomputer field has proved so ripe for publishers is precisely that different users need documentation compatible with their level of sophistication for any particular software product. It is easier to provide these different levels by separating them into individual books.

So how does all this help you to create user documentation? Is it futile even to attempt? There are those who believe there should never be a need for any printed documentation if the system is truly "user friendly." The argument is that everything a user would want to know should already have been incorporated into the program as help facilities, sophisticated error handling techniques, and the like.

Well, perhaps it's just that I've never used a truly friendly system, but I've always found a need for detailed printed documentation. In addition, I prefer printed documentation, in many instances, to that provided by the software. I would rather take some time to read over documentation before I actually try to use the system. I find that this is usually best accomplished at odd times and for short durations, in situations where using the computer would prove inconvenient, such as when riding the train.

Types of Users

I feel that users can generally be broken up into three simple groups—novices, experts, and everyone else. The novice users need the documentation in order to get started. There is usually an obvious set of questions that occurs to every novice user. Answering this set of questions can go a long way toward providing for the documentation needs of the novice user.

Highly experienced users also need printed documentation, but for a different reason. They need more details on the most advanced areas of the software so that they can get the most use out of the package. They also typically want to know every detail about every aspect of the system, and expect questions about every option to be fully answered. This is obviously much more difficult to provide than the documentation for the novice user. It requires anticipating how options might be combined to create an even more powerful tool. The novice and expert users are likely to spend a good deal of time reading the documentation.

The group that needs detailed printed documentation the least is the one that occupies the middle ground and includes the regular user. This user knows about enough features of the software and the way they work that he or she can use the application on a regular basis *without* referring to the documentation. Anytime a reference must be made, this user wants to be able to find the answer very quickly. In addition, the answer must completely specify all the options.

It is difficult to be all things to all people. How does one keep from boring the expert with trivial details while providing adequate information for the novice? How can the documentation be formatted to provide a clear narrative that guides inexperienced users, and also provide a quick, effective reference manual for experienced users?

Answering all of these questions would fill an entire book on technical writing. However, there are a number of obvious steps that can be taken to create a document that a user at any level can find useful.

General Description

The first step is to tell the (potential) user just what the system is all about. This was essentially accomplished already in the system documentation as part of the specifications. However, technical specs have a tendency to be peppered with jargon and occasionally with details that a user would not need to know. In user documentation, it is best to write your general description at the "least common denominator" level for novices. Save the details about the features until later.

Although this section will probably only be read once by the user, it is still important that the description be accurate, unambiguous, and to the point. A user immediately turned off by the documentation will not be a satisfied user. And even if those users are just your friends and neighbors, you want them to depend on your documentation—not on you—for their answers. After all, you don't want them calling you at 3 A.M. to find out how something works in your program!

One piece of information that is useful at this point is a simple statement about the intended user of the system. It may be that you developed the software with systems programmers in mind, or wrote it for your five-year-old daughter. In either case, clearly stating this up front will keep the wrong people from trying to use your software. Anyone who chooses to ignore this warning can safely be left to his or her own devices. There is nothing wrong with targeting a piece of software, just as there is nothing abnormal about targeting a book toward a particular audience. This book is just such an example. However, let your software do the targeting, not your documentation.

Hardware and Software Requirements

This, too, is essentially a rehash of what was given in the system documentation. The potential user needs this information to determine if he or she has all the proper hardware and software components for using the product. Once again, however, this information must be presented in a way that is palatable even to a novice.

Limitations

As with system documentation, the user must be warned quite quickly about any potential pitfalls of the software. This is usually highly appreciated by a potential user who is evaluating the package for possible use. It is much better to point out these "weaknesses" right up front than to let the user discover them on his or her own. That can only serve to create disgruntled users.

Detailed Description

This section of the documentation makes up the body of the text that will serve as a reference manual. This does not mean that it should be difficult to read or brimming with jargon. It does mean that the most often used information will be contained within this section.

General Start-up Instructions

Some software systems have a very simple start-up procedure: put the disk in the primary disk drive and turn on the machine. However, even in this simplest of circumstances, some ambiguity remains. A novice might not be certain which drive is the primary one. And what if the system he or she is using only has one drive? Also, does this mean that the system *must* be turned off in order to start the software? Nothing is ever as simple as it first seems.

The start-up instructions should come in three parts. First, for the novice user, include the simplest possible explanation for starting the system. It doesn't hurt to be painfully explicit in this section. This is another part of the documentation that probably will not be read more than once or twice by even a novice user. Naturally, how explicit you need to be depends on the experience level of the anticipated user. If it is likely that the user is not only a novice to this particular piece of software but also to computing itself, you will want to be very explicit in your start-up description.

Next, give a short description and example of how the regular user of this system will perform a start-up. This description has the extra benefit of beginning to educate the novice about the way the system is likely to be used. (This will be further elaborated in a later section describing a user tutorial.) For the more experienced user, this part provides the needed information for immediately using the system.

Finally, the start-up instructions should include a description of all options that can be specified during the start-up procedure. This should be a comprehensive reference that gives every last detail of the start-up, complete with examples and descriptions of feature uses. The experienced user will refer to this to discover new ways to save work by customizing the start-up procedure for a given application.

Help Facilities

Provide a brief description of how any built-in help facility is used. This should probably be brief, since the help facility itself should not be very difficult to use. A few examples of the facility here should suffice.

Input Prompts, Formats, and Options

This section will make up the bulk of the user guide. Every message the system displays that requires the user to do something must be documented here. While you are undoubtedly convinced that no one could ever misinterpret your cleverly phrased prompts, someone always does. The old saying is "No one can ever create a truly foolproof system, because fools are so ingenious." This is true in our case because the English language is so versatile. Every word has several meanings. Many words can be used as a noun, a verb, or an adjective. We have an overabundance of homonyms, synonyms, antonyms, and forty-seven other kinds of "-nyms." To make matters worse, prompt messages are usually limited in length because of the physical size of display screen format. Making up completely understandable messages is an art form similar to haiku, the Japanese poetry.

As a result, it is usually necessary to explain a prompt in more detail than can be included as part of the program itself. *Every* prompt should be included, no matter how trivial or self-explanatory it may seem. This inclusion ensures consistency, and assures the user that the prompt was not simply forgotten in the documentation.

As part of the description of the prompt, the expected input must be well documented. How inputs and their prompts are designed was discussed in Chapter 4, Human-Engineered Programming. Here we are interested in removing any remaining ambiguities from the system.

To begin with, the format of the expected input must be specified.

This should be obvious from the prompt message itself. However, since this is not always possible or practical, it must at least be presented in the user documentation.

Any options for the input must also be spelled out in complete detail. Numerous examples may be required to fully explain a complex input. There is (usually) no such thing as too many examples.

Outputs and Their Meanings

One of the biggest favors that the documentor can do a user is to provide a comprehensive list and description of the various output displays and printouts produced by a program. There is nothing more frustrating for a user than groping for the magic that would cause the program to produce a desired output. I recall quite vividly countless times when I wanted some program to produce a particular display. I wasn't sure the program was able to give me output in the desired format, but it seemed reasonable that it should. Frustration set in after I looked through manuals that only discussed inputs!

In a way, this type of documentation is like a thesaurus or an inverse dictionary (you look up the definition, then find words that match it). The intent is to aid the user in using the correct commands and selecting the right options to produce the desired display or printout.

Additional details can also be given for interpreting a particular output. Details about the meaning of the columns of a report or the parts of a chart have to be explained somewhere. This is the place.

Error Messages

Error messages have a tendency to be a real sore spot with most users. It seems that no matter how carefully the messages are thought out by the programmer, they are always less than perfectly understandable. Like the other sections of the user guide, the error message section should amplify the meaning of its subject matter.

List each message in some type of order. Many programmers like to give each message a number. Others simply use an alphabetical listing. Still others list the messages by type or keywords. No one method is the best. Providing listings in more than one format is naturally the ideal, although it is time-consuming.

Next, provide an in-depth narrative of the meaning of the message itself. Clarity is the key. Providing examples is often necessary to promote complete understanding.

Finally, some helpful hints on how to correct the error are a great idea. This might be simply telling the user to enter the information in the proper format. Sometimes, however, this will mean detailing the steps the user needs to take to backtrack in order to get to the source of the error. Most often, the solution to the problem is not easily anticipated. In these cases, at least make suggestions as to what the user might try next.

Tutorials

The purpose of a tutorial is to provide a chaperoned walk through the software in order to introduce the new user to the system. The tutorial is a way to give a detailed example of the various facilities and features of the software. The user will typically go through the tutorial once. Thereafter, he or she may occasionally use the tutorial as a reference until more familiarity with the system and the rest of the documentation is gained.

Sample Run

The easiest tutorial to provide is simply a listing of a typical run of the system. This can often be provided by directing output not only to the screen but also to the printer. Under CP/M, for example, this can be done by turning a printer toggle switch *on* by typing ^P (Control-P). After this is done everything printed on the screen is also sent to the printer, including inputs. Annotating the listing by hand should provide any additional comments that may be necessary for the user.

The problem with this technique is that it is text-oriented, not screen-oriented. If you have spent time using fancy cursor control techniques on the display, copying to the printer might not work. In addition, using graphics makes copying to the printer difficult or impossible. In these cases, you will have to produce a sample run by hand, drawing any graphics as well as possible. Another possibility would be to use photographs of screens in the documentation. This allows you to show real displays, including direct cursor addressing and graphics, even with color.

Programmed Tutorial

A much more elaborate scheme is to create a second program that leads the user through the various parts of the application program.

This would allow the user to get used to the inputs, their formats, and their options using a preset example. For instance, the Apple computer company has an excellent tutorial disk packaged with every new IIe computer system. In it the user is introduced, with text and graphics, to the keyboard of the Apple IIe. This includes describing keys, and then giving the user a chance to try those keys right away. The tutorial program monitors what the user is doing at all times so that it knows whether the user makes a mistake or does the right thing. In effect it is an entirely new program that pretends it is the computer executing the actual application program. In this way it teaches the user how to use the system.

This type of tutorial is very difficult and time-consuming to create. Unless it is handled properly, it can do more harm than good. However, it is worth noting that most reputable software companies are including programmed tutorials in their software packages. For those products that do not include a tutorial, many programmers are getting rich by writing and marketing them separately.

Index

Providing a detailed index for any document is a very tedious job. However, without this list of important topics and their page references within the user's guide, the documentation isn't much good as a reference manual. There isn't any easy way to produce an index, unfortunately. A brute force method of listing all key words or phrases and then poring over the documentation looking for references is about the only option.

However, if you have used a word processor to prepare your document, you might be able to cut the work somewhat. One way would be to create a file that contains a list of all the keywords. This file could then be matched against the document, and page references for each word in the file kept track of. This would undoubtedly take a long time to execute, but is infinitely preferable to doing the job by hand.

The User Documentation section of the appendix shows a complete set of documentation for the Compression program.

7 Software Maintenance

7.1 INTRODUCTION

Mention the term **maintenance programming** to a professional programmer and you'll get one of a number of reactions. Some will grimace in a way that will make you swear they are having a heart attack. Others will simply laugh. And watch out for the ones who will take a swing at you!

The reactions are based on the fact that every programmer has had to do his share of program maintenance, and 99.99 percent learned to hate it. They figure being a "real" programmer means spending all your time creating new programs. Only wimps do maintenance. I suppose it's like the difference between being an automotive engineer and a car mechanic.

As noted elsewhere, this reputation is not without some grounding in reality. Maintenance duties are usually shunted to the lowest ranking programmer on the staff. This programmer is lowest ranked either because of ability or lack of seniority. In either case, little respect is paid to his or her capabilities.

There are two very odd things about this situation. The first is that such an important task would be left to the least experienced personnel. That maintenance has the reputation of being unimportant causes no end of troubles. There are many maxims that suggest why this attitude is inappropriate. "There is always one more bug" is one of the most profound statements about the programming task I have ever encountered. That it is true can no longer be doubted by computer professionals. Those six words show the need to pay close attention to the duty of maintenance better than any three-day professional seminar costing $795.

Since the maintenance task can easily be shown to have grave consequences if ignored, it is doubly odd that the duty should be shunned by the best programmers. As we discussed in Chapter 5, debugging is something of a black art, calling on all the talent of the programmer. Surely, therefore, this would prove the most challenging work!

Unfortunately, ninety percent of maintenance changes are truly dull and boring. They are additions or modifications to the original specifications of the system, not the great mystery of a bug. Few people seem to enjoy this particular type of work.

The one compelling fact about maintenance is that it is necessary. The need for it is not going to disappear with the next revolution in technology. And maintenance is not the cheap part of a software project, either.

For the amateur programmer, the maintenance phase of a project is probably where the most errors are introduced into the software. If particular care is not taken, more time can be spent in this phase than in all other phases. This can happen in spite of using all the techniques outlined in this or any other book.

The reason is fairly simple. It is too easy (initially) to give up all the techniques used to this point and start right in making changes to the code. This can be incredibly seductive, especially if you have an immediate idea of what needs to be changed. Unfortunately, even if you are right (this time), doing this can destroy all the careful preparation that went into the system. The next change will be a little harder to make because part of the structure has been broken. The time after that will be even harder. By the fifth or sixth change, it will be as if you never used any of the design or implementation techniques we've discussed.

I hope you are now convinced that a little investment in the maintenance procedures will make the other techniques continue to pay off.

7·2 GETTING READY TO MAKE THE CHANGE

It is extremely important that any changes made to a fully implemented program be as carefully thought out and executed as the original program itself. Every technique that is useful for designing and implementing the program in the first place can—and should—be used when making changes. The ultimate goal is that, aside from obvious signposts to the contrary, another programmer should not be able to tell where a change was made.

There are two obvious extremes for changes to the system. The first is when there is a change in only one line of the code. In this case, little design effort is necessary. However, if this is supposed to be a fix to a bug rather than a change in the program's specifications, alarm bells should go off; fixes are seldom that easy.

In the other extreme, vast amounts of code must be added. You would think that such enormous changes would result in a virtual redesign of the entire system. However, one of the benefits of the techniques presented in this book is that they limit the need for such a thorough overhaul. If there are indeed fundamental changes in the needs of the users, then it is usually simplest to start from scratch. A fundamental flaw in the design of the system, creating a bug of such magnitude that the entire program needs to be redesigned, is unthinkable. If this is truly the case, again it would be better to start over.

Designing and Creating the New Version

Consider all parts of the original design first. This is where the extra time spent documenting the system really pays off. Unless you understand the original design, you won't be able to make changes to the system with any level of confidence. In addition, most changes will have something to do with modifying the original specifications. This means redoing any chart, diagram, or table affected by that modification in order to keep the change consistent with the original.

Be on the lookout for changes that would affect not only this particular program, but other programs within the system as well. This will most often happen when a change to a file's record format is required. Another place it can occur is when passing parameters to other modules. This all leads to Gopher Principle 2: Just when you think you have all the holes accounted for, up pops the gopher someplace else.

Most professional computing shops with large computers have automated tools for keeping track of various versions of a program. This includes differentiating between a version of a program that is being tested (called **test mode**), one that is being used to do real work (called **production mode**), and one that is not being used at all (called **archive mode**). The tool might also keep track of such details as when the last change to a version was made, and by whom. Finally, such a tool might keep track of several versions of a program within the same mode by using version numbers.

Unfortunately, such a tool is seldom available to microcomputer users, especially to the amateur. However, such a facility can make life a lot simpler when trying to keep track of various versions of numerous programs. Therefore, I suggest simulating the above features manually.

Version Mode

First, create three types of disks, one for each of the three modes (test, production, and archive). For regular use, I only use those programs on the disks marked "production." Perhaps more important, I only make modifications to programs on the test disks, never ones in production or archived. This helps avoid the nagging doubt that I'm using a program that isn't quite ready, or that I've accidentally changed a program I'm using regularly. The archive disks are used to hold any old version of programs that I want to keep around, "just in case." These disks are a kind of junk drawer. Professionals also use the archive for holding programs that are no longer in production mode because they aren't needed.

Version Numbers

Version numbers are given only to programs in test mode, so that I can try out more than one implementation of a piece of the program. When I first implement the program, I call it version 0. The version number is appended to the end of the program name. For instance, the compression program in the appendix would have been called COMPRSS0. I also include the version number in the program header comments. I continue using version 0 until the entire program has been implemented, or until I want to try a slightly different implementation as an experiment. In the latter case, I would copy the program to a new name, COMPRSS1.

When I have completed the implementation to the point where I want to begin using the program, I copy it to the production disk.

I give the program the same name as before, but without the version number. Since the version number (and other details about changes) is a part of the program header comments, I can easily find the corresponding test version if I need to.

Once I begin using the program in production mode, I do not make any further changes to the corresponding test version. If I am implementing the program somewhat incrementally, any new modules to be added will be made to a new version. This is also true whenever I need to perform maintenance on the program, whether for debugging or for adding a feature later. For instance, if I have placed COMPRSS1 into production, the next time I want to make a change to the COMPRSS program, I create a new version again by copying COMPRSS1 to COMPRSS2. I then make changes to COMPRSS2.

What good does this do? It isolates changes. If I make a disastrous change to the newest test version only, I still have older versions to fall back on. In professional shops, this is done in case a change proves faulty even after testing (i.e., in production mode). The faulty production version is **backed out** by copying the previous production version back into production. The previous production version is the last test version before the current one.

The Archive

So what happens to all these old test versions hanging around? Well, if they are important to you for some reason, copy them to an archive file and erase them from the test disk. Not every old version will be important. Versions that are essentially superseded by more recent ones need not be retained. They only take up space. Any that represent departures from the original might be worth keeping as experiments. At least the test version just previous to the current one should be archived and not simply erased, for the reason just given above—backup. A certain amount of housecleaning will be necessary from time to time to keep things from getting too cluttered.

7·3 CHANGING THE PROGRAM

Now that a change has been designed and a new version of the program generated, the actual change can be made. Once again, the techniques discussed previously for implementing a sound program should be followed with care.

Inserting New Code

This may be somewhat difficult if you are maintaining someone else's program. First, the programmer might not have followed the same guidelines or used the same techniques that you are familiar with. Second, even if the original programmer did use the same techniques you would have used, programming style is a very individual thing. Variable naming, spacing, equation construction, etc., are likely to differ greatly from programmer to programmer.

Recall from the last section that I said new code should be indistinguishable from the original code of a program. This is certainly a worthy ideal, and means that the maintenance job the next time will be no harder (depending, naturally, on the reason for the maintenance) than it was this time. It's sort of like the admonition found on signs in public parks: leave the place as clean as you found it.

Given individual style differences, however, this may not be entirely practical, especially if you don't agree with a technique or style used by the original programmer. Whenever possible, put your differences aside, unless your style blends well with the original. Don't make the code a battlefield where you wage a war against someone else's style. It will only make things worse.

Retesting

Once the change has been made, the entire system must be retested. There are several steps that need to be followed.

First, the original test data must be examined to see if it needs modifying too. Chances are it will, in order to test the new code effectively. Never depend on previously used test data to check out the new code. Ignoring this advice has caused many programmers much embarrassment when they find that their new version doesn't perform in production any better than the old one.

Second, test the program as thoroughly as if it were the first time. Look at *all* of the test results, not just the ones you assume will change. Again, the Gopher Principle will kill you if you don't.

Finally, be sure to test the entire system, not just the one program that might have changed. Even when there is certainty that a new change cannot possibly affect the interface between two programs, problems can sneak in. For instance, imagine inadvertently deleting a line from the code during the editing of the source code. What if that line of code was supposed to write a particular record out to a file that another program used? It happens all too easily.

7·4 KEEPING THE DOCUMENTATION UP TO DATE

This part is the simplest of all the maintenance tasks. Maybe that is the reason it is too often forgotten. It is probably also related to the lack of respect most computer professionals seem to have for documentation: "Oh, I believe in good documentation. Just don't expect me to write any."

Every programmer, however, has had more than one experience with trying to use documentation that didn't match the program. This probably happened not because of some error when the documentation was originally written, but because the program was changed and the documentation wasn't. Never forget that once the integrity of the documentation becomes suspect, the documentation becomes worthless. It is a little like suspecting your spouse has been cheating on you. Once such a suspicion has been entertained, the trust is usually irreparably broken.

The first source of error in documentation is the program listing itself. The internal comments of the program need to be updated to include any change. Any new modules or blocks of code should be given the same documenting treatment as any original code. This includes header comments as well as line comments. What good does it do the maintenance programmer in the future if the comments say the code does exactly the opposite of what it really does?

Also, be certain to update the version number and to add comments to explain what changes were made, when, and by whom. These last details are also helpful on every line of code added or changed. In this way, the programmer who wrote those lines can be identified by any future programmer who has a question. In addition, the lines associated with each version of the program can be traced.

The system documentation should be simple to modify. Just add or replace figures and descriptions within the package. Again, be sure to include version numbers, dates, and names in order to provide a trail of changes. Finally, be sure to create an error report for the system documentation.

The user documentation probably won't need to be changed for most modifications to the system. This should certainly be true for any fixes of bugs. In the case of changes or additions to the features of the system, you will probably only need to write up the details of these new modifications. However, be careful that all related items within the user guide are changed. For example, adding a feature probably requires adding text to the descriptions of the inputs, the outputs, the error messages generated, and the index. Finally, the tutorial would have to be modified to include the new feature.

Appendix: A Compression Program— Complete Documentation

SYSTEM DOCUMENTATION

Program Name:	COMPRESS
File Name:	COMPRSS.BAS

General Description:
The purpose of this program is to remove all comments from a BASIC program. There are two main benefits to this. First, the resulting code usually requires substantially less memory in order to execute, since in BASIC comments take up real space. Also, since a well-documented program can have as many lines of comments as executable code, the result will often be a program that is only half the size of the original. This can be crucial when dealing with large programs and relatively small amounts of available memory (anything less than about 64K bytes) in the computer. Second the program will execute faster. This is because, even though comments are not "executable" instructions, an interpreted language such as BASIC must actively ignore the comments every time it encounters them. This takes significant time if there are many comments throughout the code.

Hardware Requirements:

Type of Computer: anything that will run Microsoft BASIC, such as the IBM PC or DEC Rainbow 100

Memory Requirement: 12K bytes, plus BASIC interpreter

Mass Storage Required: at least one floppy disk drive

Software Requirements:

Operating System: anything that will run Microsoft BASIC, e.g., CP/M 80/86 for the DEC Rainbow 100

Language: Microsoft BASIC. Could be translated to other versions by substituting appropriate mechanisms for file handling and string manipulation. This includes OPEN, CLOSE, and PRINT# instructions, and INSTR, MID$, LEFT$, and LEN string functions.

Files: Input—the file that contains a BASIC program that will be compressed (the comments are removed). This file must be in ASCII format, not tokenized form.

Output—the file that contains the compressed form of the input file.

Design:

H-Chart: See Figure 1.

Algorithms: The method is to read in one line of the source program at a time. This line is then searched for a comment. It is assumed that a line contains only one comment, since the search ends immediately when a REM or ' is encountered. If a comment is found, the line is truncated beginning with the start of the comment (the R or '). If this truncation results in an empty line, because the entire line was a comment, the entire line is removed from the target program by not writing it out. If the rest of the line was not empty, then the truncated line is written out. If there was no comment on the line, then nothing happens and the line is written to the target file. See Figure 2 for detailed algorithms for the routines.

Special Techniques:

1. There is the possibility that a comment beginning with REM will contain one or more single quotation marks in the following comment, or that a comment beginning with ' will contain a word beginning with the three letters REM. In order to allow this situation, the source line is searched twice, first for REM, then for a single quote. It is assumed that the comment begins with the one found first on the line. It is possible that these searches turn up neither REM nor '. In this case, there is simply no comment in that line.

2. If a line of the program is only a comment, with no executable instructions on it, then the entire line must be kept out of the target file. Otherwise there will be lines in the target file that contain a line number, but nothing else. This will cause an error when an attempt is made to load the file. To avoid this, after the comment is truncated from the line, the resulting line is searched for blanks. If, after the line number, the entire line is blank, the line is set to null and is not written to the target program file.

Error Handling:

1. The user is allowed to change the file names entered for the source and target file names.

2. The file name of the source program must be different from the file name of the target file. If they are not different, the user must reenter both file names.

Restrictions and Special Considerations:

1. The source program file must have been saved in ASCII form using the following save command:
   ```
   SAVE "⟨file-name⟩" ,A
   ```

2. The source and target files must have different names.

3. The three letters REM cannot appear anywhere in a string literal within the program. For instance,
   ```
   100 PRINT "The REM statement will be deleted"
   ```
   will be truncated to
   ```
   100 PRINT "The
   ```

4. The single quotation mark (') cannot appear anywhere in a string literal within the program. For instance,
   ```
   100 PRINT "Enter 'Y' or 'N'"
   ```
   will be truncated to
   ```
   100 PRINT "Enter
   ```

5. Tabs cannot precede a comment if the entire line is a comment. For instance,

   $$100\ \underbrace{\quad}_{\text{(tab)}}\underbrace{\quad}_{\text{(tab)}}\underbrace{\quad}_{\text{(tab)}}\ \text{'comment}$$

   will result in the line

   $$100\ \underbrace{\quad}_{\text{(tab)}}\underbrace{\quad}_{\text{(tab)}}\underbrace{\quad}_{\text{(tab)}}$$

   being entered into the target program file. This will cause an error when an attempt is made to load the target program.

Future Extensions:

1. Remove restrictions (2) and (3) above. This might be done by checking, after a comment indicator (REM or ') is found in a line, whether the indicator is surrounded with double quotes. This can be determined by searching the string again for ".

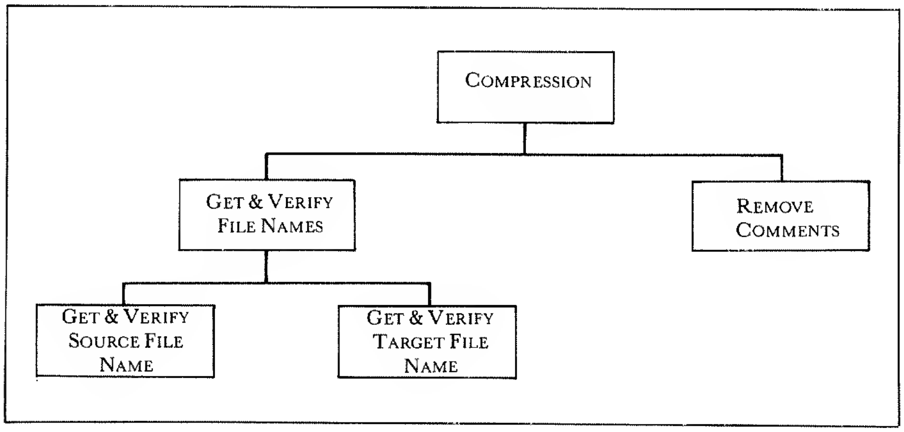
2. Remove restriction (4) above. This can be done in the Delete-Empty-Line routine by searching for both a blank and a tab symbol (ASCII 9). If the line is entirely made up of a series of blanks and tabs, then the line is empty.

Test Results:

Test Data: See Figure 3.

Test Results: See Figure 4.

FIGURE 1



```
Compress.
BEGIN
  OUTPUT instructions;
  Get-and-Verify-File-Names(Source, Target);
  OPEN Source File;
  OPEN Target File;
  Remove-Comments;
  Close Files
END.

Get-and-Verify-File-Names(Source, Target).
BEGIN
  REPEAT
    Get-and-Verify-Source-File-Name(Source);
    Get-and-Verify-Target-File-Name(Target);
    IF Source file name = Target file name THEN
      OUTPUT ("Error")
  UNTIL Source file name <> Target file name
END;

Remove-Comments.
BEGIN
  WHILE more lines in Source file DO
  BEGIN
    INPUT a Line from the Source file;
    search the Line for a comment;
    IF there is a comment in the Line THEN
      truncate the Line beginning with the comment;
    IF the Line is not empty THEN
      OUTPUT the Line to the Target file
  END
END;

Get-and-Verify-Source-File-Name(Source).
BEGIN
  REPEAT
    INPUT Source file name from the user
  UNTIL user says name is OK
END;

Get-and-Verify-Target-File-Name(Target).
BEGIN
  REPEAT
    INPUT Target file name from the user
  UNTIL user says name is OK
END;
```

FIGURE 2: **Algorithms**

FIGURE 3: **Test data**

```
10 'This is a test file for the compress program
20 REM this is line 1
30 'This is line 2
40 REM will this work if ' is included?
45 PRINT "This line should lose its comment"    'comment
50 'Will this work if REM is in this line?
60 '
70 REM
80 PRINT "This should have all remark statements removed"
90 END
```

```
45 PRINT "This line should lose its comment"
80 PRINT "This should have all remark statements removed"
90 END
```

FIGURE 4: **Test results**

PROGRAM LISTING

```
1  '##############################################################
10 ' Program Name:    COMPRESS
11 ' File Name:       COMPRSS.BAS
12 ' Version:         1
13 '
14 ' Author:          Blaise W. Liffick
15 '
16 '     (c) 1984 Blaise W. Liffick, for noncommercial use only
17 '
18 ' The purpose of the program is to remove all comments from
19 ' another program.  This makes the size of this source program
20 ' much smaller, since comments take up real space in a BASIC
21 ' program.  The only expected inputs are the name of the source
22 ' file, and the name of a file that will contain the compressed
23 ' version when the program is done.
24 '
25 ' Input file:     SOURCE$, file that contains the original
26 '                      source program. Name is supplied by user
27 '                 TARGET$, file that contains the compressed
28 '                          version of the source program.  Name
29 '                          is supplied by the user.
30 '
31 ' Variable Dictionary:
32 '   SOURCE$ - The name of the Source program file
33 '   SOURCE13$, SOURCE15$, SOURCE19$ - Formal parameters for
34 '      various subroutines for the variable SOURCE$
35 '   TARGET$ - The name of the Target program file
36 '   TARGET13$, TARGET19$ - Formal parameter for various
37 '      subroutine for the variable TARGET$
38 '   RESPONSE$, RESPONSE15$, RESPONSE19$ - Any response by the
39 '      user
40 '   LINE23$ - The line from the Source file that is currently
41 '      being processed
42 '   LINE27$, LINE30$ - Formal parameters for various
43 '      subroutines for the variable LINE23$
44 '   REMLOC27% - The position of the first occurrence of of REM
45 '      in the line currently being processed
46 '   QUOTELOC27% - The position of the first occurrence of a
47 '      single quotation mark in the line currently being
48 '      processed
49 '   POSITION30% - A pointer to blanks in the line being
50 '      processed
51 '   BLANK30$ - A single blank constant
71 '
72 ' Modification History:
73 '    Version 0, 10/26/84:   formal parameters for subroutines
74 '    Version 1, 10/27/84:   routine 3010, delete empty lines
99 '##############################################################
```

(continued)

```
110 PRINT
120 PRINT "*****************************************************"
130 PRINT "*                                                   *"
140 PRINT "*                Compression Program                *"
150 PRINT "*                                                   *"
160 PRINT "*             (c) 1984 Blaise W. Liffick            *"
170 PRINT "*                                                   *"
180 PRINT "*****************************************************"
190 PRINT
200 PRINT
210 PRINT "This program removes all remark statements from a BASIC"
220 PRINT "program.  This compresses the size of a program down to"
230 PRINT "a minimum.  This is useful when a program becomes too"
240 PRINT "large to fit into memory.  Since comments take up space"
250 PRINT "in a BASIC program, removing them should make the whole"
260 PRINT "program smaller.  Obviously, this is only helpful if"
270 PRINT "the remaining code is smaller than available memory."
290 PRINT
300 '
310 ' repeat
320   PRINT
330   INPUT "Hit RETURN to continue.",RESPONSE$
340 IF RESPONSE$ <> "" GOTO 320         'until RETURN entered
350 '
360 PRINT
370 PRINT
380 PRINT
390 PRINT "***** WARNING *****"
400 PRINT
410 PRINT "The program file you want to compress must have been"
420 PRINT "saved in ASCII form, not 'tokenized' form.  You must"
430 PRINT "have saved the file using the command"
440 PRINT
450 '
460 ' CHR$(34) is a double quote mark.  We can't just PRINT "
470 '
480 PRINT "             SAVE ";CHR$(34);"filename";CHR$(34);",A"
490 '
492 '>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
493                    'no input parameters for routine 1310
500 GOSUB 1310         'Get-and-Verify-File-Names
502                    'set output parameters for routine 1310
504 SOURCE$ = SOURCE13$
506 TARGET$ = TARGET13$
510 '<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
520 'open the files
530 '
540 OPEN "I",#1,SOURCE$
550 OPEN "O",#2,TARGET$
560 '
561 '>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
562                    'no input parameters for routine 2310
570 GOSUB 2310         'Remove-Comments
580                    'no output parameters for routine 2310
585 '<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
586 '
590 CLOSE              'all files
600 PRINT
```

```
 610 PRINT
 620 PRINT "The file ";TARGET$;" now contains the compressed"
 625 PRINT "version of ";SOURCE$
 630 PRINT
 640 PRINT "***** End of Compression Program *****"
 650 GOTO 9999   'end of program
1300 '=============================================================
1301 '          Get-and-Verify-File-Names
1302 '
1303 ' Output parms:  SOURCE13$, file name of source program
1304 '                TARGET13$, file name of compressed program
1305 '
1306 ' This routine gets the source and target file names.
1307 '-------------------------------------------------------------
1308 'repeat
1309 '>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
1310   GOSUB 1510       'Get-and-Verify-Source-File-Name
1312                    'set output parameter for routine 1510
1314   SOURCE13$ = SOURCE15$
1315 '<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
1316 '
1317 '>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
1318                    'set input parameter for routine 1910
1319   SOURCE19$ = SOURCE13$
1320   GOSUB 1910       'Get-and-Verify-Target-File-Name
1322                    'set output parameter for routine 1910
1324   TARGET13$ = TARGET19$
1325 '<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
1326 '
1330 IF SOURCE13$ = TARGET13$ THEN
        PRINT:
        PRINT "***** The file names cannot be the same.  *****":
        PRINT "***** Reenter. *****":
        GOTO 1310
1340 'until names are different
1350 '
1360 'names are now different
1370 '
1399 RETURN
1500 '=============================================================
1501 '          Get-and-Verify-Source-File-Name
1502 '
1503 ' Output parameter:  SOURCE15$, file name of source program
1504 '
1505 ' This routine gets the name of the program to be compressed
1506 ' from the user.
1507 '-------------------------------------------------------------
1509 'repeat
1510   PRINT
1520   PRINT
1530   PRINT "What is the complete name of the program file you";
1535   PRINT " want compressed?"
1540   PRINT
1550   INPUT "Name: ",SOURCE15$
1560   'repeat
1570     PRINT
1580     PRINT "The name of the program file to be";
1585     PRINT " compressed is ";SOURCE15$
```

(continued)

```
1590     PRINT
1600     INPUT "Is this name correct? (Y/N)",RESPONSE15$
1610   IF (RESPONSE15$ = "Y") OR (RESPONSE15$ = "y") OR
          (RESPONSE15$ = "N") OR (RESPONSE15$ = "n") GOTO 1690
1620     ' invalid response
1630     PRINT
1640     PRINT "***** Invalid response.  Enter Y or N. *****"
1650   GOTO 1570                'until response is valid
1660 '
1670 'response is valid
1680 '
1690 IF (RESPONSE15$ = "N") OR (RESPONSE15$ = "n") GOTO 1510
1700 'until name is valid
1799 RETURN
1900 '===========================================================
1901 '          Get-and-Verify-Target-File
1902 '
1903 ' Input parameter:  SOURCE19$, file name of source program
1904 ' Output parameter: TARGET19$, file name of compressed program
1905 '
1906 ' This routine gets the name of the file that will contain
1907 ' the compressed version of the source program.
1908 '-----------------------------------------------------------
1909 'repeat
1910   PRINT
1920   PRINT
1930   PRINT "What is the name of the program file for the"
1940   PRINT "compressed version of ";SOURCE19$;"?"
1950   PRINT
1960   INPUT "Name: ",TARGET19$
1970   'repeat
1980     PRINT
1990     PRINT "The compressed program file name is ";TARGET19$
2000     PRINT
2010     INPUT "Is this correct? (Y/N)",RESPONSE19$
2020     IF (RESPONSE19$ = "Y") OR (RESPONSE19$ = "y") OR
            (RESPONSE19$ = "N") OR (RESPONSE19$ = "n") GOTO 2100
2030       'invalid response
2040       PRINT
2050       PRINT "***** Invalid response.  Enter Y or N. *****"
2060     GOTO 1980       'until response is valid
2070 '
2080 'response is valid
2090 '
2100 IF (RESPONSE19$ = "N") OR (RESPONSE19$ = "n") GOTO 1910
2110 'until name is valid
2120 '
2130 'target file name is now valid
2160 '
2199 RETURN
2300 '===========================================================
2301 '          Remove-Comments
2302 '
2303 ' No parameters
2304 '
2305 ' This routine locates and eliminates any comments from a
2306 ' line of the source program.  If the entire line was a
```

```
2307 ' comment, it is not included in the target file at all.
2308 '------------------------------------------------------------
2310 WHILE NOT EOF(1)          'while more statements in source file
2320 LINE INPUT#1,LINE23$      'get next statement from source file
2330 '
2340 'search for REM or single quote
2350 '
2372 '>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
2373                    'set input parameters for routine 2710
2375 LINE27$ = LINE23$
2377 '
2380 GOSUB 2710          'Search-for-Comment
2382                    'set output parameter from routine 2710
2384 LOCATN23% = LOCATN27%
2386 '<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
2390 '
2400 'LOCATN23% holds location of first comment indicator, if there
2410 'is one.  This is in case we have a single quote in a REM, or
2420 'use the letters REM in a comment when we have a single quote
2430 'starting the comment. The comment in line 2420 is an example.
2440 '
2450 'If there is a comment, truncate the line
2460 '
2470 IF LOCATN23% <= 0 GOTO 2540          'no comment in this line
2475   LINE23$ = LEFT$(LINE23$,LOCATN23%-1)   'truncate the comment
2490   '
2491   '>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
2492                    'set input parameter for routine 3010
2494   LINE30$ = LINE23$
2496   '
2500   GOSUB 3010        'Delete-Empty-Line
2502                    'set output parameter from routine 3010
2504   LINE23$ = LINE30$
2506   '<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
2510 '
2520 ' if the resulting line was deleted, don't write it to the
2530 ' file.
2540 '
2550 IF LINE23$ <> "" THEN PRINT#2,LINE23$
2560 WEND
2599 RETURN
2700 '============================================================
2701 '          Search-for-Comment
2702 '
2703 ' Input parameter:  LINE27$, line to be searched for a comment
2705 ' Output parameter: LOCATN27%, location of first remark
2706 '                     indicator in the line, REM or ', if any
2707 '
2708 ' This routine locates the position of a remark in a line.
2709 '------------------------------------------------------------
2710 REMLOC27% = INSTR(LINE27$,"REM")         'look for REM
2720 QUOTELOC27% = INSTR(LINE27$,"'")         'look for single quote
2725 '
2726 'there is no comment in the line if both are zero
2727 '
2730 IF (REMLOC27% = 0) AND (QUOTELOC27% = 0) THEN
         LOCATN27% = 0:
         GOTO 2899
```

(continued)

```
2734 '
2735 'either REMLOC27% or QUOTELOC27% is not 0, or both
2736 '
2740 IF REMLOC27% = 0 THEN LOCATN27% = QUOTELOC27%: GOTO 2899
2750 IF QUOTELOC27% = 0 THEN LOCATN27% = REMLOC27%: GOTO 2899
2760 '
2770 'neither is zero, so take the smallest
2780 '
2790 IF REMLOC27% < QUOTELOC27% THEN
          LOCATN27% = REMLOC27%
     ELSE
          LOCATN27% = QUOTELOC27%
2899 RETURN
3000 '=============================================================
3001 '          Delete-Empty-Line
3002 '
3003 ' Input parm:  OUTLINE30$, line to be searched for all blanks
3004 ' Output parm: OUTLINE30$
3005 '
3006 ' This routine determines if what is left after eliminating
3007 ' the remark from this line is all blanks.  If so, the entire
3008 ' line is eliminated by making it null.
3009 '-------------------------------------------------------------
3010 BLANK30$ = CHR$(32)                'one blank
3020 POSITION30% = INSTR(LINE30$,BLANK30$)
3030 WHILE (POSITION30% <= LEN(LINE30$)) AND
          (MID$(LINE30$,POSITION30%,1) = BLANK30$)
3040   POSITION30% = POSITION30% + 1
3050 WEND
3060 '
3070 'if the entire line was blank, delete the line
3080 '
3090 IF POSITION30% > LEN(LINE30$) THEN LINE30$ = ""
3199 RETURN
9999 END
```

USER DOCUMENTATION

Program: COMPRESS
File Name: COMPRSS.BAS

General Description:
The purpose of this program is to remove all comments from a BASIC program. There are two main benefits to this. First, the resulting code usually requires substantially less memory in order to execute, since in BASIC comments take up real space. Also, since a well-documented program can have as many lines of comments as executable code, the result will often be a program that is only half the size of the original. This can be crucial when dealing with large programs and relatively small amounts

of available memory (anything less than about 64K bytes) in the computer. Second the program will execute faster. This is because, even though comments are not "executable" instructions, an interpreted language such as BASIC must actively ignore the comments every time it encounters them. This takes significant time if there are many comments throughout the code.

Hardware Requirements:

Type of Computer: anything that will run Microsoft BASIC, such as the IBM PC or DEC Rainbow 100

Memory Requirement: 12K bytes, plus BASIC interpreter

Mass Storage Required: at least one floppy disk drive

Software Requirements:

Operating System: anything that will run Microsoft BASIC, e.g., CP/M 80/86 for the DEC Rainbow 100

Language: Microsoft BASIC. Could be translated to other versions by substituting appropriate mechanisms for file handling and string manipulation. This includes OPEN, CLOSE, and PRINT# instructions, and INSTR, MID$, LEFT$, and LEN string functions.

Files: Input—the file that contains a BASIC program that will be compressed (the comments are removed). This file must be in ASCII format, not tokenized form.

Output—the file that contains the compressed form of the input file.

Restrictions and Special Considerations:

1. The source program file must have been saved in ASCII form using the following save command:
   ```
   SAVE "⟨file-name⟩",A
   ```

2. The Source and Target files must have different names.

3. The three letters REM cannot appear anywhere in a string literal within the program. For instance,
   ```
   100 PRINT "The REM statement will be deleted"
   ```
   will be truncated to
   ```
   100 PRINT "The
   ```

4. The single quotation mark (') cannot appear anywhere in a string literal within the program. For instance,

   ```
   100 PRINT "Enter 'Y' or 'N'"
   ```

   will be truncated to

   ```
   100 PRINT "Enter
   ```

5. Tabs cannot precede a comment if the entire line is a comment. For instance,

   100 $\underbrace{\quad}_{\text{(tab)}}\underbrace{\quad}_{\text{(tab)}}\underbrace{\quad}_{\text{(tab)}}$ 'comment

   will result in the line

   100 $\underbrace{\quad}_{\text{(tab)}}\underbrace{\quad}_{\text{(tab)}}\underbrace{\quad}_{\text{(tab)}}$

   being entered into the target program file. This will cause an error when an attempt is made to load the target program.

Start-up Instructions:

This program is very simple to operate. The user has minimal interaction with the program once it is started. The user will be asked to supply the names of two files. The first file, called the Source file, is the name of the file that contains the program you want compressed. This program must have been saved in BASIC using the ASCII format option. The general form of the command to do this is

```
SAVE "⟨file-name⟩",A
```

Whatever you used for ⟨file-name⟩ when issuing this command is the name you will supply to the COMPRESS program for the Source file.

The other file used in this program is called the Target file. This is the file that will contain the compressed version of the program that is in the Source file. When the program is done, the Source file will still contain the complete program, with all its comments, while the Target file will contain just the executable statements of the program.

Step 1: Save the Source program in ASCII format.

Step 2: Load the compression program using

```
LOAD "COMPRSS.BAS"
```

Step 3: Type RUN.

Step 4: The program will then ask you to supply the Source and Target file names. The only restriction here is that THE SOURCE AND TARGET FILES MUST HAVE DIFFERENT NAMES.

Inputs:

The user will receive the following prompts for input:

"Hit RETURN to continue.":
The only response by the user that is accepted is to strike the RETURN or ENTER key.

"What is the name of the program file you want compressed?"
The user should enter the name of the file containing the program that is to have its comments removed. The program name should be exactly as it was specified when the program was SAVED, and should include the .BAS extension. For example, REMTEST.BAS.

"What is the name of the program file for the compressed version of ⟨source-file-name⟩?"
The user should enter the name of the target file that is to contain the compressed version of the source file. ⟨source-file-name⟩ is the name of the file you entered in response to the prompt for the source file name.

"Is this correct? (Y/N)"
The user is given an opportunity to change either of the file names entered. If the name is correct, the user should enter a Y, followed by RETURN. If the name is not correct (e.g., it was misspelled), the user should enter an N. No other response is accepted.

Error Messages:

"***** *Invalid response. Enter Y or N.* *****"
Meaning: The user has entered a response that the program did not recognize.
Correction: The user must enter either a Y or an N only.

"***** *The file names cannot be the same. Reenter.* *****"
Meaning: The user entered the exact same name for both the Source program file and the Target program file. The program will not allow the names to be the same, however.
Correction: Enter a different name for the Target file than was specified for the Source file.

Sample Run:

The following is an example of how this compression program executes. The items that are underlined denote entries made by the user.

```
*****************************************************
*                                                   *
*                Compression Program                *
*                                                   *
*             (c) 1984 Blaise W. Liffick            *
*                                                   *
*****************************************************


This program removes all remark statements from a BASIC
program.  This compresses the size of a program down to
a minimum.  This is useful when a program becomes too
large to fit into memory.  Since comments take up space
in a BASIC program, removing them should make the whole
program smaller.  Obviously, this is only helpful if
the remaining code is smaller than available memory.


Hit RETURN to continue.


***** WARNING *****

The program file you want to compress must have been
saved in ASCII form, not 'tokenized' form.  You must
have saved the file using the command

              SAVE "filename",A


What is the complete name of the program file you want compressed?

Name: REMTEST.BSA

The name of the program file you want compressed is REMTEST.BSA

Is this name correct? (Y/N)  B

***** Invalid response.  Enter Y or N. *****

The name of the program file you want compressed is REMTEST.BSA

Is this name correct? (Y/N)  N


What is the complete name of the program file you want compressed?

Name: REMTEST.BAS

The name of the program file you want compressed is REMTEST.BAS

Is this name correct? (Y/N)  Y


What is the name of the program file for the
compressed version of REMTEST.BAS?
```

```
Name: REMTST.BAS

The compressed program file name is REMTST.BAS

Is this correct? (Y/N)  N


What is the name of the program file for the
compressed version of REMTEST.BAS?

Name: REMTEST.BAS

The compressed program file name is REMTEST.BAS

Is this correct? (Y/N)  Y

***** The file names cannot be the same.  *****
*****  Reenter. *****


What is the complete name of the program file you want compressed?

Name: REMTEST.BAS

The name of the program file you want compressed is REMTEST.BAS

Is this name correct? (Y/N)  Y


What is the name of the program file for the
compressed version of REMTEST.BAS?

Name: REMOUT.BAS

The compressed program file name is REMOUT.BAS

Is this correct? (Y/N)  Y


The file REMOUT.BAS now contains the compressed
version of REMTEST.BAS

***** End of Compression Program *****
```

Bibliography

1. Allman, E., and M. Stonebraker. "Observations on the Evolution of a Software System." *Computer*, Vol. 15, No. 6 (June 1982). [1]
2. Ashcroft, E., and Z. Manna. "The Translation of 'go to' Programs to 'while' Programs." Originally appeared in *Proceedings of the 1971 IFIP Congress*. North-Holland Publishing Co., 1972. Reprinted in *Classics in Software Engineering*. E. Yourdon, editor. New York: Yourdon Press, 1979. [2]
3. Bailey, T., and K. Lundgaard. *Program Design with Pseudocode*. Monterey, CA: Brooks/Cole Publishing Co., 1983. [2]
4. Baker, F. "System Quality Through Structured Programming." Originally appeared in the *AFIPS Proceedings of the 1972 Fall Joint Computer Conference*, Vol. 41. AFIPS Press, 1972. Reprinted in *Classics in Software Engineering*. E. Yourdon, editor. New York: Yourdon Press, 1979. [2]
5. Balzer, R., R. Cheatham, and C. Green. "Software Technology in the 1990s: Using a New Paradigm." *Computer*, Vol. 16, No. 11 (November 1983). [1]
6. Basili, V., and B. Perricone. "Software Errors and Complexity: An Empirical Investigation." *Communications of the ACM*, Vol. 27, No. 1 (January 1984). [5]
7. Bass, L., and R. Bunker. "A Generalized User Interface for Applications Programs." *Communications of the ACM*, Vol. 24, No. 12 (December 1981). [4]
8. Bayman, P., and R. Mayer. "A Diagnosis of Beginning Programmers' Misconceptions of BASIC Programming Statements." *Communications of the ACM*, Vol. 26, No. 9 (September 1983). [3]
9. Be, K. "Human-Computer Interaction." *Computer*, Vol. 15, No. 11 (November 1982). [4]
10. Bell, R. "Monte Carlo Debugging: A Brief Tutorial." *Communications of the ACM*, Vol. 26, No. 2 (February 1983). [5]

11. Bentley, J. "A Case Study in Applied Algorithm Design." *Computer*, Vol. 17, No. 2 (February 1984). [2]

12. Bergland, G. "A Guided Tour of Program Design Methodologies." *Computer*, Vol. 14, No. 10 (October 1981). [2]

13. Berk, T., L. Brownston, and A. Kaufman. "A Human Factors Study of Color Notation Systems for Computer Graphics." *Communications of the ACM*, Vol. 25, No. 8 (August 1982). [4]

14. Berns, G. "Assessing Software Maintainability." *Communications of the ACM*, Vol. 27, No. 1 (January 1984). [7]

15. Boehm, B. "Software Engineering." Originally appeared in *IEEE Transactions on Computers*, Vol. C-25, No. 12 (December 1976). Reprinted in *Classics in Software Engineering*. E. Yourdon, editor. New York: Yourdon Press, 1979. [1]

16. Boehm, B., and T. Standish. "Software Technology in the 1990s: Using an Evolutionary Paradigm." *Computer*, Vol. 16, No. 11 (November 1983). [1]

17. Bruce, R. *Software Debugging for Microcomputers*. Reston, VA: Reston Publishing Co., 1980. [5]

18. Bryan, W., S. Siegel, and G. Whiteleather. "Auditing Throughout the Software Life Cycle: A Primer." *Computer*, Vol. 15, No. 3 (March 1982). [1]

19. Carey, T. "User Differences in Interface Design." *Computer*, Vol. 15, No. 11 (November 1982). [4]

20. Chapman, D. "A Program Testing Assistant." *Communications of the ACM*, Vol. 25, No. 9 (September 1982). [4,5]

21. Crawford, C. "The Atari Tutorial; Part 10: Human Engineering." *BYTE*, Vol. 7, No. 6 (June 1982). [4]

22. Dahl, O., E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming*. New York: Academic Press, 1972. [2,3]

23. DeSanctis, G., and J. Courtney. "Toward Friendly User MIS Implementation." *Communications of the ACM*, Vol. 26, No. 10 (October 1983). [4]

24. Dijkstra, E. W. *A Discipline of Programming*. Englewood Cliffs, NJ: Prentice-Hall, 1976. [2,3]

25. ———. "Programming Considered as a Human Activity." Originally appeared in *Proceedings of the 1965 IFIP Congress*. North-Holland Publishing Co., 1965. Reprinted in *Classics in Software Engineering*. E. Yourdon, editor. New York: Yourdon Press, 1979. [2,3]

26. ———. "Go To Statement Considered Harmful." Originally appeared as a letter to the editor in *Communications of the ACM*, Vol. 11, No. 3 (March 1968). Reprinted in *Classics in Software Engineering*. E. Yourdon, editor. New York: Yourdon Press, 1979. [2,3]

27. ———. "Structured Programming." Reprinted in *Classics in Software Engineering*. E. Yourdon, editor. New York: Yourdon Press, 1979. [2,3]

28. ———. "The Humble Programmer." Originally appeared in *Communications of the ACM*, Vol. 15, No. 10 (October 1972). Reprinted in *Classics in Software Engineering*. E. Yourdon, editor. New York: Yourdon Press, 1979. [1]

29. Dillon, R. "Human Factors in User-Computer Interaction: An Introduction." *Behavior Research Methods and Instrumentation*, Vol. 15, No. 2 (1983). [4]

30. Donaldson, J. "Structured Programming." Originally appeared in *Datamation*, Vol. 19, No. 12 (December 1973). Reprinted in *Classics in Software Engineering*. E. Yourdon, editor. New York: Yourdon Press, 1979. [2,3]

31. Dwyer, B. "A User-Friendly Algorithm." *Communication of the ACM*, Vol. 24, No. 9 (September 1981). [2]

32. Dwyer, T., and M. Critchfield. *BASIC and the Personal Computer.* Reading, MA: Addison-Wesley, 1978. [2,3,6]

33. ———. *A Bit of BASIC.* Reading, MA: Addison-Wesley, 1980. [2,3,6]

34. ———. *CP/M and the Personal Computer.* Reading, MA: Addison-Wesley, 1983. [2]

35. ———. *You Just Bought a Personal What?* Peterborough, NH: BYTE Books, 1980. [2,3,6]

36. Efe, K., C. Miller, and K. Hopper. "The Kiwinet-Nicola Approach: Response Generation in a User-Friendly Interface." *Computer*, Vol. 16, No. 9 (September 1983). [4]

37. Eisenhardt, M. "A User-Friendly Software Environment for the Novice Programmer." *Communications of the ACM*, Vol. 26, No. 12 (December 1983). [4]

38. Eishoff, J., and M. Marcotty. "Improving Computer Program Readability to Aid Modification." *Communications of the ACM*, Vol. 25, No. 8 (August 1982). [6,7]

39. Fleckenstein, W. "Challenges in Software Development." *Computer*, Vol. 16, No. 3 (March 1983). [1]

40. Gauss, E. "The 'Wolf Fence' Algorithm for Debugging." *Communications of the ACM*, Vol. 25, No. 11 (November 1982). [5]

41. Gilb, T., and G. Weinberg. *Humanized Input.* Cambridge, MA: Winthrop Publishers, Inc., 1977. [4]

42. Giller, W., and S. Pollack. *An Introduction to Engineered Software.* New York: Holt, Reinhart and Winston, 1982. [1,2,3,5]

43. Girill, T., and C. Luk. "DOCUMENT: An Interactive, Online Solution to Four Major Document Problems." *Communications of the ACM*, Vol. 26, No. 5 (May 1983). [6]

44. Glass, R. *Software Reliability Guidebook.* Englewood Cliffs, NJ: Prentice-Hall, 1979. [5]

45. Gonnet, G., and F. Tompa. "A Constructive Approach to the Design of Algorithms and Their Data Structures." *Communications of the ACM*, Vol. 26, No. 11 (November 1983). [2]

46. Grosberg, J. "Replacing the Term 'Software Maintenance'." *Computer*, Vol. 17, No. 1 (January 1984). [7]

47. Harrison, W., K. Magel, R. Kluczny, and A. DeKock. "Applying Software Complexity Metrics to Program Maintenance." *Computer*, Vol. 15, No. 9 (September 1982). [7]

48. Hearn, A. "Top-Down Modular Programming." In *Program Design*, Blaise W. Liffick, editor. Peterborough, NH: BYTE Books, 1978. [1]

49. ———. "Some Words About Program Design." In *Program Design*, Blaise W. Liffick, editor. Peterborough, NH: BYTE Books, 1978. [1]

50. Heines, J. *Screen Design Strategies for Computer-Assisted Instruction.* Bedford, MA: Digital Press, 1984. [4]

51. Heite, N., and L. Heite. "Breaking the Jargon Barrier." *BYTE*, Vol. 6, No. 7 (July 1982). [4]

52. Hopkins, M. "A Case for the GOTO." Originally appeared in *Proceedings of the 25th National ACM Conference,* ACM, August 1972. Reprinted in *Classics in Software Engineering*. E. Yourdon, editor. New York: Yourdon Press, 1979. [3]

53. Howard, J. "What Is Good Documentation?" *BYTE*, Vol. 6, No. 3 (March 1981). [6]

54. Howden, W. "Contemporary Software Development Environments." *Communications of the ACM*, Vol. 25, No. 5 (May 1982). [4]

55. Jacky, J., and I. Kalet. "A General Purpose Data Entry Program." *Communications of the ACM*, Vol. 26, No. 6 (June 1983). [4]

56. Jacob, R. "Using Formal Specifications in the Design of a Human-Computer Interface." *Communications of the ACM*, Vol. 26, No. 4 (April 1984). [4]

57. Kamins, S., and M. Waite. *Apple Backpack*. Peterborough, NH: BYTE Books, 1982. [4]

58. Kernighan, B., and P. Plauger. "Programming Style: Examples and Counterexamples." Originally appeared in *ACM Computing Surveys*, Vol. 6, No. 4 (December 1974). Reprinted in *Classics in Software Engineering*. E. Yourdon, editor. New York: Yourdon Press, 1979. [3,6]

59. Knuth, D. "Structured Programming with go to Statements." Originally appeared in *Current Trends in Programming Methodology*. ACM, 1974. Reprinted in *Classics in Software Engineering*. E. Yourdon, editor. New York: Yourdon Press, 1979. [3]

60. Kruesi, E. "The Human Engineering Task Area." *Computer*, Vol. 16, No. 11 (November 1983). [4]

61. Leclerc, Y., S. Zucker, and D. Leclerc. "A Browsing Approach to Documentation." *Computer*, Vol. 15, No. 6 (June 1982). [6]

62. Ledin, G., and V. Ledin. *The Programmer's Book of Rules*. Belmont, CA: Wadsworth, Inc., 1979. [3,4,6]

63. Levy, M. "Modularity and the Sequential File Update Problem." *Communications of the ACM*, Vol. 25, No. 6 (June 1982). [1]

64. Lew, A. "On the Emulation of Flowcharts by Decision Tables." *Communications of the ACM*, Vol. 25, No. 12 (December 1982). [3]

65. Lewis, T. *Software Engineering for Micros*. Rochelle Park, NJ: Hayden Book Co., 1979. [1]

66. Linger, R., H. Mills, and B. Witt. *Structured Programming: Theory and Practice*. Reading, MA: Addison-Wesley, 1979. [1,2,5]

67. Long, L. *Data Processing Documentation and Procedures Manual*. Reston, VA: Reston Publishing Co., 1979. [6]

68. Luehrman, A. "Structured Programming in BASIC." *Creative Computing*, (May, June, and July 1984). [3]

69. Mackey, K., and T. Slesnick. "A Style Manual for Authors of Software." *Creative Computing*, (August 1982). [4]

70. McCracken, D. "Revolution in Programming: An Overview." Originally appeared in *Datamation*, Vol. 19, No. 12 (December 1973). Reprinted in *Classics in Software Engineering*. E. Yourdon, editor. New York: Yourdon Press, 1979. [1]

71. Miara, R., J. Musselman, J. Navarro, and B. Schneiderman. "Program Indentation and Comprehensibility." *Communications of the ACM*, Vol. 26, No. 11 (November 1983). [1,3,6]

72. Miller, E., and G. Lindamood. "Structured Programming: Top-Down Approach." Originally appeared in *Datamation*, Vol. 19, No. 12 (December 1973). Reprinted in *Classics in Software Engineering*. E. Yourdon, editor. New York: Yourdon Press, 1979. [2,3]

73. Mizuno, Y. "Software Quality Improvement." *Computer*, Vol. 16, No. 3 (March 1983). [5]

74. Morland, D. "Human Factors Guidelines for Terminal Interface Design." *Communications of the ACM*, Vol. 26, No. 7 (July 1983). [4]

75. Mosak, A. "Structured Programming Can Be Applied to Microprocessors—Even by Novices." *IEEE Micro*, Vol. 2, No. 1 (February 1982). [3]

76. Munson, J. "Software Maintainability: A Practical Concern for Life-Cycle Costs." *Computer*, Vol. 14, No. 11 (November 1981). [7]

77. Myers, G. *The Art of Software Testing*. New York: John Wiley & Sons, 1979. [5]

78. Nagin, P., and H. Ledgard. *BASIC With Style*. Rochelle Park, NJ: Hayden Book Co., 1978. [3,6]

79. Neidetz, M. "Flowchart 'Filosophy'." *Computer*, Vol. 15, No. 11 (November 1982). [3]

80. Norman, D. "Design Rules Based on Analysis of Human Error." *Communications of the ACM*, Vol. 26, No. 4 (April 1984). [2,4,5]

81. Page-Jones, M. *The Practical Guide to Structured Systems Design*. New York: Yourdon Press, 1980. [1,2]

82. Parkin, A. *Systems Analysis*. Cambridge, MA: Winthrop Publishers, 1980. [1,2]

83. Parnas, D. "On the Criteria to Be Used in Decomposing Systems into Modules." Originally appeared in *Communications of the ACM*, Vol. 5, No. 12 (December 1972). Reprinted in *Classics in Software Engineering*. E. Yourdon, editor. New York: Yourdon Press, 1979. [1]

84. Parnas, E. "A Technique for Software Module Specification with Examples." *Communications of the ACM*, Vol. 15, No. 5 (May 1972). [1]

85. Pfaff, G., H. Kuhlmann, and H. Hanusa. "Constructing User Interfaces Based on Logical Input Devices." *Computer*, Vol. 15, No. 11 (November 1982). [4]

86. Philippakis, A., and L. Kazmier. *Program Design Concepts*. New York: McGraw-Hill, 1983. [1,2]

87. Pressman, R. *Software Engineering: A Practitioner's Approach*. New York: McGraw-Hill, 1982. [1,2,3]

88. Richardson, G., C. Butler, and J. Tomlinson. *A Primer on Structured Design*. New York: Petrocelli Books, 1980. [1,2,3]

89. Rockart, J., and L. Flannery. "The Management of End User Computing." *Communications of the ACM*, Vol. 26, No. 10 (October 1983). [4]

90. Roman, G., M. Stucki, W. Ball, and W. Gillet. "A Total System Design Framework." *Computer*, Vol. 7, No. 5 (May 1984). [1]

91. Ryan, H. "End-User Game Plan." *Datamation*, Vol. 29, No. 12 (December 1983). [4]

92. Schneider, G., S. Weingart, and D. Perlman. *Programming and Problem Solving With Pascal*, 2d ed. New York: John Wiley & Sons, 1982. [3]

93. Schneider, M., and J. Thomas. "The Humanization of Computer Interfaces." *Communications of the ACM*, Vol. 26, No. 4 (April 1984). [4]

94. Schneiderman, B. "Designing Computer System Messages." *Communications of the ACM*, Vol. 25, No. 9 (September 1982). [4]

95. ———. *Software Psychology*. Cambridge, MA: Winthrop Publishers, Inc., 1980. [4]

96. Semprevivo, P. *Systems Analysis*, 2d ed. Chicago, IL: Science Research Associates, Inc., 1982. [1,2]

97. Shooman, M. *Software Engineering*. New York: McGraw-Hill, 1983. [1,5]

98. Simpson, H. "A Human-Factors Style Guide for Program Design." *BYTE*, Vol. 7, No. 4 (April 1982). [4]

99. Skvarcius, R. *Problem Solving Using Pascal*. Boston: PWS Publishers, 1984. [3]

100. Sobell, M. "Structured Programming in BASIC." *BYTE*, Vol. 7, No. 1 (January 1982). [3]

101. Soloway, E., J. Bonar, and K. Ehrlich. "Cognitive Strategies and Looping Constructs: An Empirical Study." *Communications of the ACM*, Vol. 26, No. 11 (November 1983). [3]

102. Stevens, W., G. Myers, and L. Constantine. "Structured Design." Originally appeared in *IBM Systems Journal*, Vol. 13, No. 2 (May 1974). Reprinted in

Classics in Software Engineering. E. Yourdon, editor. New York: Yourdon Press, 1979. [1,2]

103. Thayer, R., A. Pyster, and R. Wood. "Validating Solutions to Major Problems in Software Engineering Project Management." *Computer*, Vol. 15, No. 8 (August 1982). [1,5]

104. Vessey, I., and R. Weber. "Some Factors Affecting Program Repair Maintenance: An Empirical Study." *Communications of the ACM*, Vol. 26, No. 2 (February 1983). [7]

105. Vile, R. *Apple II Programmer's Handbook*. Englewood Cliffs, NJ: Prentice-Hall, 1982. [3]

106. Vitalari, N., and G. Dickson. "Problem Solving for Effective Systems Analysis: An Exploration." *Communications of the ACM*, Vol. 26, No. 11 (November 1983). [1]

107. Weinberg, G. *The Psychology of Computer Programming*. New York: Van Nostrand Reinhold Co., 1971. [3,4]

108. ———. *Rethinking Systems Analysis and Design*. Boston: Little, Brown and Company, 1982.

109. Weiser, M. "Programmers Use Slices When Debugging." *Communications of the ACM*, Vol. 25, No. 7 (July 1982). [5]

110. Wile, D. "Program Developments: Formal Explanations of Implementations." *Communications of the ACM*, Vol. 26, No. 11 (November 1983). [3]

111. Williams, G. "Applied Structured Programming . . . and How to Use It." In *Program Design*. Blaise W. Liffick, editor. Peterborough, NH: BYTE Books, 1978. [3]

112. ———. "Is This Really Necessary? A First Look At Design Techniques." *BYTE*, Vol. 6, No. 3 (March 1981). [2]

113. ———. "Structured Programming and Structured Flowcharts." *BYTE*, Vol. 6, No. 3 (March 1981). [3]

114. Wirth, N. *Algorithms + Data Structures = Programs*. Englewood Cliffs, NJ: Prentice-Hall, 1976. [2]

115. Wirth, N. "On the Composition of Well-Structured Programs." Originally appeared in *Computing Surveys*, Vol. 6, No. 4 (December 1974). Reprinted in *Classics in Software Engineering*. E. Yourdon, editor. New York: Yourdon Press, 1979. [2]

116. Wirth, N. "Program Development by Stepwise Refinement." *Communications of the ACM*, Vol. 14, No. 4 (April 1971). [1,2]

117. Wooldridge, S. *Systems and Programming Standards*. New York: Petrocelli/Charter, 1977. [7]

118. Wulf, W. "A Case Against the GOTO." Originally appeared in *Proceedings of the 25th National ACM Conference*. ACM, August 1972. Reprinted in *Classics in Software Engineering*. E. Yourdon, editor. New York: Yourdon Press, 1979. [3]

119. Yourdon, E. *Managing the Structured Techniques*. Englewood Cliffs, NJ: Prentice-Hall, 1979. [1]

120. Zave, P. "The Operational Versus the Conventional Approach to Software Development." *Communications of the ACM*, Vol. 27, No. 2 (February 1984). [1]

121. Ziegler, C. *Programming System Methodologies*. Englewood Cliffs, NJ: Prentice-Hall, 1983. [1]

122. ———. "Writing Your Own." *Business Computer Systems*, June 1984. [6]

123. ———. *De Re Atari*. Sunnyvale, CA: Atari, Inc., 1981. [4]

Index

> $17.95 FPT USA

The Software Developer's Sourcebook

Blaise W. Liffick

Foreword by
Esther Dyson

Learning to program is easy; learning to develop sophisticated, high-quality software is not. Although thousands of books can teach you how to write code, here is the first book to teach you the secrets of professional programmers. *The Software Developer's Sourcebook* is an invaluable reference to the sophisticated process and advanced techniques that software designers apply to their efforts.

Everyone knows that having a good idea for a software package simply isn't enough. Taking a step-by-step approach to the development of software, *The Software Developer's Sourcebook* will show you how to:

- Design and write structured programs
- Apply human engineering principles to your software
- Use efficient testing and debugging procedures and techniques
- Write effective system and user documentation
- Maintain your program

The Software Developer's Sourcebook is the first stylebook of program development written for the average programmer. From systems analysis to designing testing schemes, here is the source for turning good ideas into top-notch software.

Blaise W. Liffick, formerly an editor and senior writer for *Byte,* is assistant professor of computer science at Millersville University, Millersville, Pennsylvania, and a consultant for microcomputer and educational computing.

Cover design by Marshall Henrichs

ISBN 0-201-11771-1

Addison-Wesley Publishing Company, Inc.